NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MODELLING, VISIBILITY TESTING AND PROJECTION OF AN ORTHOGONAL THREE DIMENSIONAL WORLD IN SUPPORT OF A SINGLE CAMERA VISION SYSTEM

by

James Earl Stein

March 1992

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE (Include Security Classification)
MODELLING, VISIBILITY TESTING AND PROJECTION OF AN ORTHOGONAL THREE DIMENSIONAL WORLD IN SUPPORT OF A SINGLE CAMERA VISION SYSTEM (U)

12. PERSONAL AUTHOR(S)
Stein, James Earl

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 01/90 TO 03/92	March 1992	180

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Three Dimensional Modelling, Visibility Testing, Perspective Projection

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Proper interpretation of the environment is essential for mission planning and navigation of an autonomous mobil robot. An on board vision system may provide the most useful raw data. This work develops part of a vision syste for the Naval Postgraduate School's mobile robot, Yamabico-11. Accurately modeling the robot's environment is im perative to support position verification and path planning. The decision to use an extended two dimensional mode an orthogonal wire-frame representation, is discussed. Additionally, to support pattern matching, a package of graph ic routines, utilizing traditional algorithms and an innovative sweep algorithm (to determine line segment visibility has been developed. This work demonstrates that an asymmetric model is appropriate to represent a three dimensiona environment in support of vision interpretation for mobile robots.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Yutaka Kanayama	22b. TELEPHONE (Include Area Code): (408) 646-2095; 22c. OFFICE SYMBOL: CS/Ka

Approved for public release; distribution is unlimited

MODELLING, VISIBILITY TESTING AND PROJECTION OF AN ORTHOGONAL THREE DIMENSIONAL WORLD IN SUPPORT OF A SINGLE CAMERA VISION SYSTEM

by

James Earl Stein
Lieutenant, USN
B.S. of Computer Science, Pennsylvania State University, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

Proper interpretation of the environment is essential for mission planning and navigation of an autonomous mobile robot. An on board vision system may provide the most useful raw data. This work develops part of a vision system for the Naval Postgraduate School's mobile robot, Yamabico-11. Accurately modeling the robot's environment is imperative to support position verification and path planning. The decision to use an extended two dimensional model, an orthogonal wire-frame representation, is discussed. Additionally, to support pattern matching, a package of graphic routines, utilizing traditional algorithms and an innovative sweep algorithm (to determine line segment visibility), has been developed. This work demonstrates that an asymmetric model is appropriate to represent a three dimensional environment in support of vision interpretation for mobile robots.

I. INTRODUCTION

A. BACKGROUND

Research to develop accurate sensors for robots continues world wide. Much of this research is focused on developing robot vision systems. As a sensor, vision is intuitively desirable since it parallels our own sight and a large amount of passive data resides in a single image.

Yamabico-11 is an autonomous, mobile robot which is under continuous development by students and faculty at the Naval Postgraduate School (NPS), Monterey Ca. The robot's operating environment is the fifth floor of an academic building, Spanagel Hall. We are interested in expanding Yamabico's sensor system to include visual image interpretation.

B. CURRENT STATE OF YAMABICO

Yamabico is currently fitted with a set of ultrasonic sonar transducers, which act as its primary sensors. The sonar array has a limited range of about four meters (imposed by the hardware) and returns from surfaces which are not perpendicular to the transmitting transducer can be very poor.

Position within the operating environment is provided at the start of a mission and a dead-reckoning (DR) system estimates current position by tracking rotation of the two drive wheels. This estimate remains accurate so long as no wheel slippage occurs.

It is desirable to develop an additional sensor system with a greater range and the ability to verify the estimated DR position.

C. DESIRED VISION SYSTEM

Yamabico is being developed to operate in a purely manmade, indoor environment. For ease of implementation, this operating environment is assumed to be orthogonal. In light of the many limitations that arise from having an active sonar array as the sole sensor, we wish to add an on board vision system to Yamabico. The assumption of an orthogonal environment greatly reduces the complexity of designing this system, since only straight lines need be considered.

The vision system will receive input from a single RGB video camera. The system will primarily be used to verify the DR estimate of Yamabico's position in the environment. To meet this need a model of the environment must be maintained on board. The DR position and course will be used to determine if the image provided by the camera coincides with what is expected from the model. Discrepancies between the model and camera views will be used to calculate and correct for any errors which occur in the DR tracking system. Figure 1.1 illustrates the interrelation of major vision system components.

D. PROBLEM STATEMENT

Three major components are necessary to implement a vision system, specifically a model of the environment with supporting functions, image processing facilities and pattern matching facilities. The thrust of this work will be directed at developing an appropriate model and it's required support functions. These functions will be needed to ensure proper storage and retrieval of model data. Another student, Kevin Peterson will

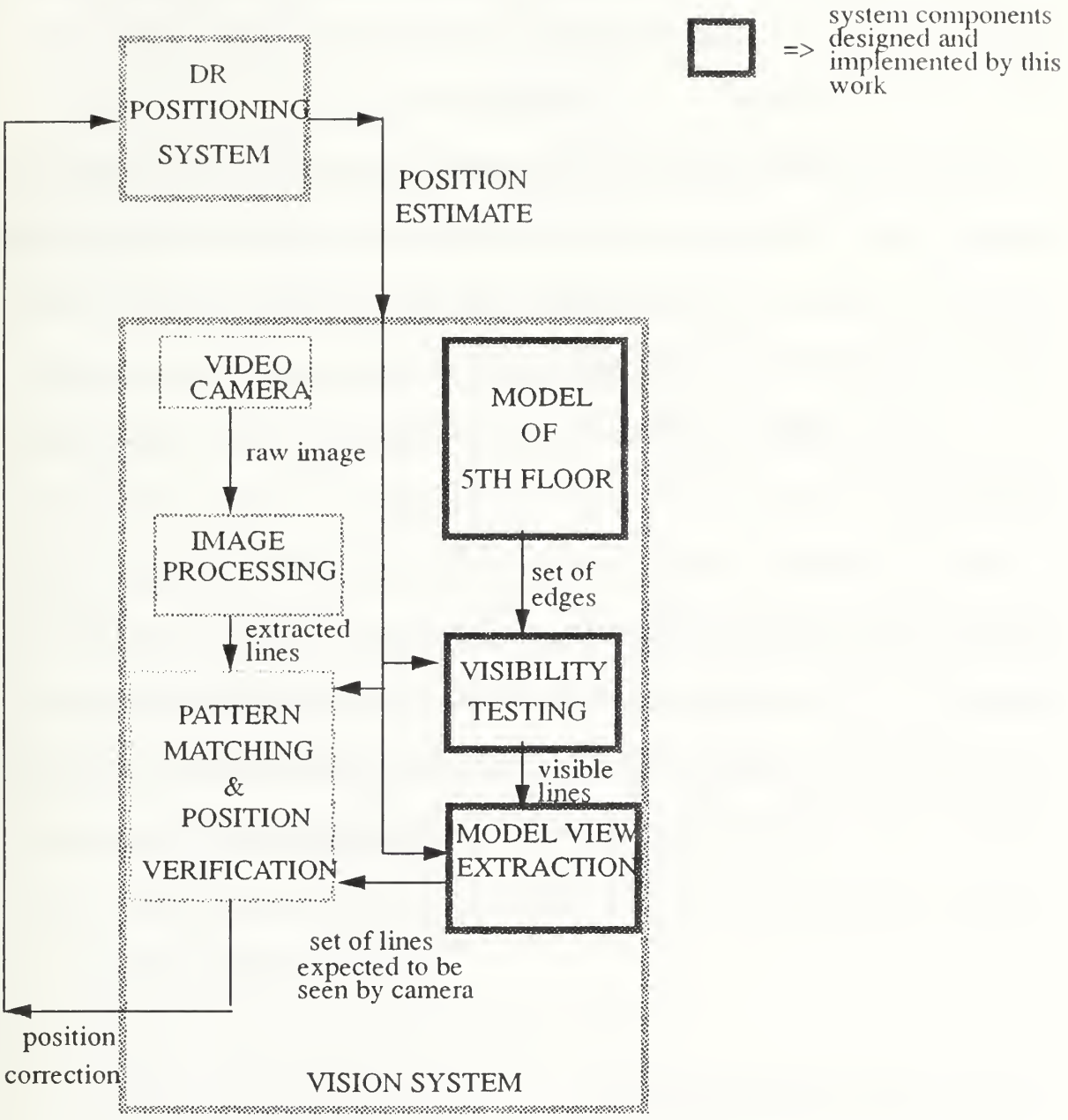


Figure 1.1

Block diagram of vision system components and data flow.

develop the processing facilities needed to extract straight line features from a camera image and the pattern matching facilities to match the model view to the camera image.

E. FOCUS OF WORK

The decision of what type of model to use must be considered carefully. The model should support the current as well as the future functionality of Yamabico. The representation used should eliminate redundant data storage wherever possible, and still provide simple real-time recovery of information. Likewise, support routines for model construction and determining the set of lines comprising a view in the model must be developed.

With the above items in mind, we are primarily concerned with supporting the vision system's pattern matching facilities. These will be used to compare a view from the camera with a view from the model. The DR position estimate and orientation in the horizontal plane will be used to extract the view, that Yamabico should see, from the model. Pattern matching between the two views will determine if Yamabico's DR estimate is correct, and if it is not, should provide the angle and magnitude of error.

F. RESEARCH METHODOLOGY

Prior to developing any algorithms and deciding on a model, review of available literature pertaining to machine vision was conducted. A vast amount of theoretical and experimental research is being conducted to develop useful machine vision systems. Some

theoretical works are attempting to find a generalized solution, but virtually all projects are designed to solve a very specific vision problem. Our literature review attempted to find research with similar goals and to analyze their degree of success while gaining exposure to some common approaches and theories.

The next step was to develop a simple surface model of Yamabico's environment. This hands on experience helped identify needs and problems which had to be addressed in the final model decision. With these factors in mind and a good idea of what information was required to support pattern matching and Yamabico's other functions, a model was chosen. Routines for model construction were completed and data, from blue prints and physical measurements, was entered concurrent with the coding of routines for data retrieval.

While work on the model took place, Kevin Peterson designed the image processing routines needed to extract a set of lines from a camera image. The final position verification system will use lines extracted from the model and the camera image as input to determine the error in Yamabico's DR position estimate.

G. THESIS ORGANIZATION

Subsequent chapters will describe the following:

1. findings of the literature review
2. the model decision and design
3. visibility checking of model lines
4. graphic support functions for the model

5. implementation, conclusions and ideas for future work

Chapter VII is a user's manual to help the reader and those working on Yamabico understand the code developed during this research. A complete listing of the final code can be found in appendix A.

II. LITERATURE REVIEW

A. RESEARCH APPROACH

Two major objectives were pursued while conducting the literature survey. The first was to gain insight into the history of computer vision. Text books which described the general goals, mathematical foundations and traditional approaches to computer vision were reviewed. The general idea was to gain a sound understanding of vision fundamentals and exposure to commercial systems currently in use.

The second objective involved reviewing more recent material to gain insight into the current state of research in the field of computer vision and model representation. We were specifically looking for projects with similar goals to our own. Identification of such projects is necessary to avoid duplicating the work of others and can also yield performance evaluations of methods which may be useful in our own project.

B. GENERAL REVIEW

As stated by [Ref. 1], there are three major phases of computer vision: choosing a digital image representation, processing the image data and analyzing the processed results to guide an application. The thrust of our work tends towards the last of these phases. Modeling the world and extracting views are essential to support interpretation of the processed camera image.

Since our work will interface with the image processing portions of the Yamabico vision system, it was valuable to review the feature extraction techniques presented in [Ref. 2]. Shirai presented traditional feature extraction methods using Hough's transform

and region merging. In contrast, [Ref. 3] describes straight line finding which combines gradient image analysis and Sobel operators to recognize edges which exhibit a sudden intensity change. [Ref. 4] detailed out the mathematical basis for image understanding, concentrating on methods of applying statistics, i.e. using error probabilities to determine the threshold values for decision rules.

Most of the projects reviewed were either targeting a different type of environment for their vision system or were employing some type of sensor other than the single camera which we are implementing on Yamabico. In the case of the Carnegie Mellon Navlab, [Ref. 5], the environment is an outdoor scene with a pronounced road to follow, while the on board vision system relies heavily on active range sensors. We find many vision systems which utilize range data from either sonar or laser range finders. The interpretation of stereo images, as in [Ref. 6] and [Ref 2], appears to be a major research avenue also.

The representation used for storing information about a system's environment is greatly influenced by the nature of that environment and is generally tailored to some degree to support the target applications or sensors. [Ref. 7] uses midpoint representation for lines which are tracked between successive images with a Kalman filter. A similar parametric representation is presented in [Ref 8] for the same type of tracking. In [Ref 9] a more complex method of geometric hashing calculates affine transformations of individual objects for storage and pattern matching.

Once a set of lines has been extracted from the camera image and a model of the environment has been created, we are interested in extracting a view from that model and

using pattern matching to compare the two. [Ref 10] provides all the basic graphic techniques which are needed to extract a view from a model. It covers the application of rotation, translation and scaling matrix operations on three dimensional objects. Projection of three dimensional objects onto a two dimensional plane along with the associated clipping algorithms are explained.

C. CONCLUSIONS

Although the scope of this work is only to develop part of Yamabico's vision system, it is desirable to gain a firm understanding of the entire target system. This is true, to some extent, when developing any software system. A preconceived notion of the functional units required by the final system may allow for smoother interface construction and lends validity to design decisions. Both these factors can minimize changes late in the software life cycle. Review of the reference material has provided us with useful background data on the entire vision process, from image digitization to pattern matching.

Our vision system will incorporate a single camera and matching will need to be done between the processed image and a view from an orthogonal world. The first stage of development involves selecting an appropriate model representation for this world. Some methods for representation of lines are mentioned above. These representations have been developed to directly support pattern matching. One of these may prove useful for depicting a view from the model, but they not entirely appropriate for storing the entire world model. Several papers, i.e. [Ref. 11], [Ref. 12] and [Ref. 13], present innovative

vision analysis methods, but neglect to describe the model representation schemes used to support them. Many papers, not appearing in the bibliography, were also reviewed in an attempt to inspect different representation methods, but again a significant lack of reference to underlying models was encountered.

The search for representation methods was not totally futile. Most references which were books vice papers did present some methods. In [Ref. 2] the uses of generalized cylinders, B-splines, extended gaussian images (EGI), and geometric models are introduced. We also find a description of surface models in [Ref. 1], and terrain maps in [Ref. 5]. Since we are attempting to model a straight line world generalized cylinders, terrain maps and B-splines are not appropriate. EGI and surface model representations seem to lend themselves more directly to storage of our world and will be considered in the model decision.

There is a substantial amount of information available concerning computer vision. Many basic ideas have been successfully implemented in industrial applications, apparently proving the value of vision research. Over the last decade or so, researchers have continued to modify these basic methods in an attempt to improve efficiency and expand the problem domain. Additionally, many interesting, original techniques have been developed to solve a variety of specific problem. This vast research effort has swamped the field with a wide variety of claims and theories regarding these techniques and their performance potential. Unfortunately, very few authors provide the reader with proof, either empirical or mathematical, of their claims. Although many forums are available to disseminate information on new techniques, there is a distressing lack of

expert review and analysis of them. This results in a huge body of unsubstantiated work, each author claiming that a particular method is reasonable but that theirs is better. As a new researcher, it was particularly daunting to sift through this ever increasing body of information with no expert guidance from any organization within the field.

Due to the previous comments and differences between our goals and those of the authors reviewed, it is unlikely that this work will take up where another has left off. Hopefully the future will see the formation of a review committee of experts, within the field of computer vision, which can investigate and report on the variety of claims being made.

III. THE MODEL

A. REQUIRED USES

The ultimate goal in Yamabico's development is to maintain a single, on board model of the fifth floor which can be used to support all functions which may need environment information. This model should also be flexible enough to accommodate the data storage and retrieval needs of future Yamabico systems. The following requirements were taken into consideration as items which must be supported by the model:

1. Accurate modelling of the three dimensional (3D) orthogonal world, fifth floor Spanagel Hall.

2. Obstacle identification for the two dimensional (2D) path planning routines used to control Yamabico's movements.

3. Position verification and displacement calculations through pattern matching of key features viewed from an assumed position.

4. Relatively easy to use interface for entry and modification of 3D world structures.

B. SURFACE MODEL

To gain a more thorough understanding of the problems and complexity which could be encountered in modeling the 5th floor of Spanagel Hall, we first developed a simple surface model of the world. Although not trivial, the interface program is fairly straight forward. The user is allowed to enter polyhedral shapes into the world. Each of these shapes receives tags which specify if the shape is fixed in the model and if it is an enclosure or an obstacle. Each polyhedron is defined by a list of surface polygons (walls,

floor and ceiling). These polygons are listed in arbitrary order and are tagged with a boolean value labeling them as convex or concave.

During data entry, it became apparent that the interface required expansion to allow for shape modifications and deletions. It also became evident that the user was wasting effort when adding similar objects (such as doors) at different locations in the model. The interface was modified to allow for the necessary deletions and modifications. In the final version a user is able to store a shape he has created to a disk file and then add copies of that object into his world model at any point while specifying the degree of rotation desired about the z axis. Likewise, when the user is done working with the model, they can store it to a binary file (from which it may later be retrieved).

A high degree of inefficiency is present in the above simple surface model. Many edges are common to more than one surface polygon. For each of these, the vertices which define the edge are redundantly stored in each polygon's vertex list. When we consider that virtually all edges in a three dimensional model are shared between at least two surfaces and each vertex between three edges, it becomes very evident that a different representation is needed to help minimize storage requirements (especially for large, complex models).

C. TAILORING TO THE APPLICATION

In addition to the question of redundancy, we ask ourselves how a representation might be tailored to our target application. In this case, we are concerned with supporting the movement of Yamabico-11. Since this robot is of constant size and for all practical

purposes only moves in two dimensions (along a horizontal plane), we can indeed tailor our thinking somewhat. Yamabico currently solves path planning problems through use of a two dimensional model and associated algorithms. Although work to perfect path planning under differing conditions continues, it seems reasonable to assume the basic approach (which does very well) will change little. Furthermore, the addition of a third dimension to the problem does not significantly alter it. The only added burden in a 3D environment is to ensure the projection of all obstacles occurring along the height of Yamabico into the 2D plane that is being used for path planning. When approached in this fashion we find a certain asymmetric quality among the three dimensions. Specifically, since Yamabico travels through the x-y plane we recognize that the importance of explicit representations in x and y information is greater than that of z information. This perspective has lead us to discard the symmetric surface model and move on to an asymmetric 'two dimensional plus' (2D+D) model.

D. THE 2D+D MODEL

Bearing all of the afore mentioned in mind, it is important to remember that the prime importance of developing a model is to ensure accurate representation of the world. Information in the representation may be explicit within the chosen data structures, or it may be implicit. In either case all information must be readily available or reconstructible.

The 2D+D model will be represented in computer memory by a group of data structures joined by pointers into linked lists. Basically, the model consists only of

horizontal surfaces and vertical edges. With this representation we can support two dimensional path planning and all edges of an orthogonal world can be reconstructed for pattern matching and graphic display. Each horizontal surface is represented by a polygon which is defined by a list of vertices. We maintain the convention used in 2D path planning, where polygons which are obstacle components have vertices listed counter clockwise while those of enclosures are in clockwise order. The vertical edges of objects in the model will be represented by pointers between vertices. Linked lists will be used extensively, and doubly linked list will be implemented where necessary to enhance performance. The following types of structures are used to describe a three dimensional world: WORLD, POLYHEDRON, POLYGON, VERTEX and INSTANCE.

The full definitions of these structures can be found in Appendix A (pg A-3) while Figure 3.1 illustrates the organization of them within memory. One parent WORLD is used to reference a 2D+D world model. For Yamabico, this will generally be the current model of the 5th floor of Spanagel Hall. This WORLD will point to a list of objects. Each object is designated as an obstacle or an enclosure and is represented by a POLYHEDRON structure. This structure in turn will point to a list of defining horizontal POLYGONs and a list of INSTANCEs. The list of POLYGONs will be in a local coordinate system and each INSTANCE will indicate a location in the world where a copy of that POLYHEDRON resides.

Each horizontal surface will be classified as a floor or a ceiling. Groups of surfaces which make up a polyhedron will be linked together and sorted by their z values. Floors will have a list of pointers to associated ceilings which, in the case of an obstacle, bound

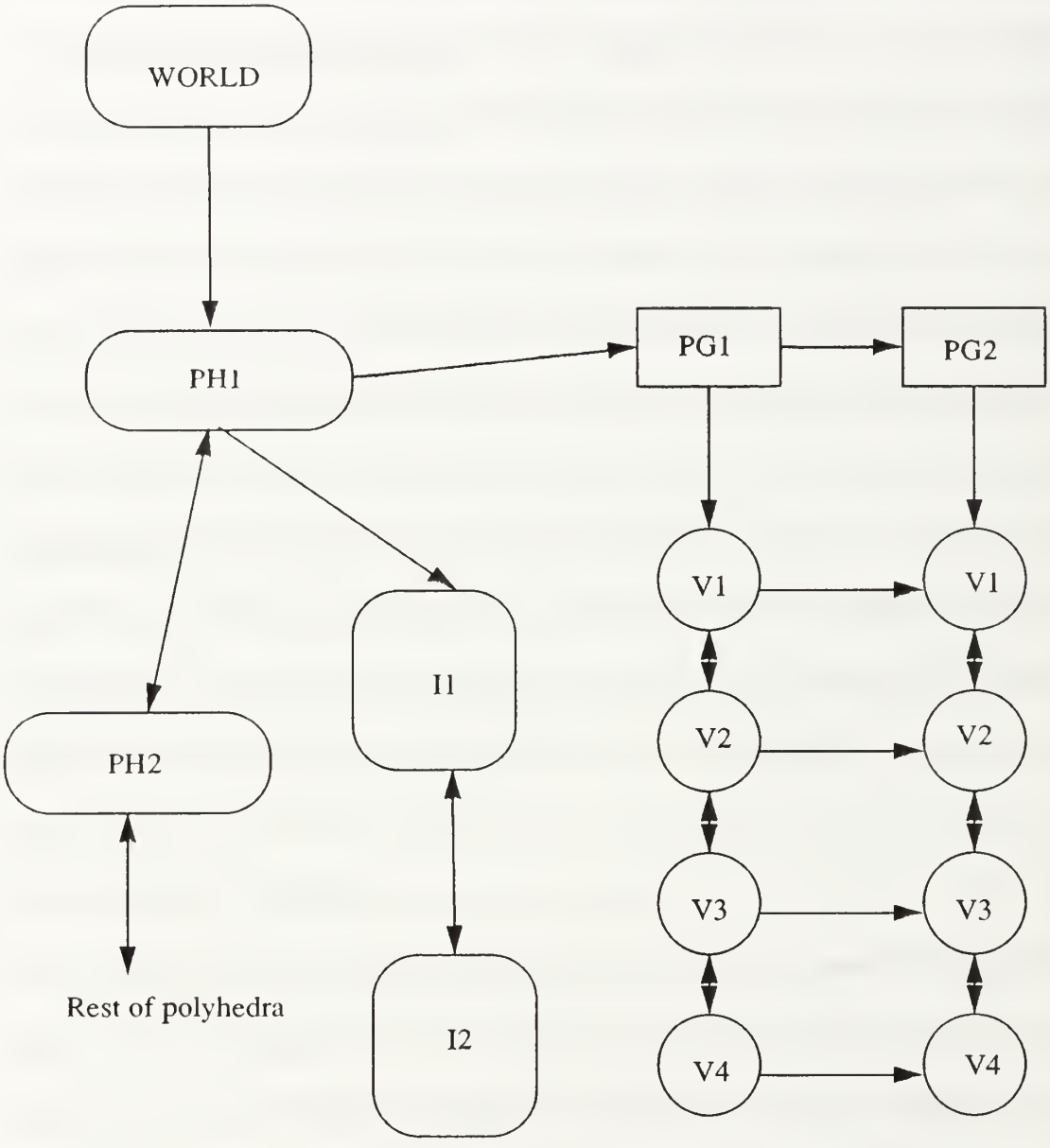


Figure 3.1
Illustration of data structure relationships in the
2d+ model (2nd polyhedron empty)
PH = polyhedron
PG = polygon in horizontal plane
I = instance of polyhedron

a solid column above that floor, and in the case of an enclosure bound a column of free space above the floor. Figure 3.2a illustrates a simple object with one floor and ceiling while 3.2b shows an obstacle with multiple ceilings.

Vertex x and y information for each POLYGON is found in a doubly linked list of VERTEX structures. Each vertical edge is represented as a pointer from the VERTEX of one POLYGON to a VERTEX in a different POLYGON. Since we are representing an orthogonal world, each vertex of a polygon will be allowed only one vertical outgoing edge. These vertical edges may be from floor to ceiling, floor to floor, or ceiling to ceiling. Note that these pointers should only be in the direction of increasing z value to avoid redundantly storing edge information (in both directions).

As previously mentioned, each POLYHEDRON will also point to a list of INSTANCEs. An instance will have a label and define where in the world coordinate system (x,y,z) a copy of that POLYHEDRON resides, what vertex in the object is to act as the pivot point for rotation about the z axis and the number of degrees the POLYHEDRON is rotated about the z axis (ROT). Storage of INSTANCES in this fashion reduces the storage requirements of the model and also readily supports movement of objects, i.e. opening doors.

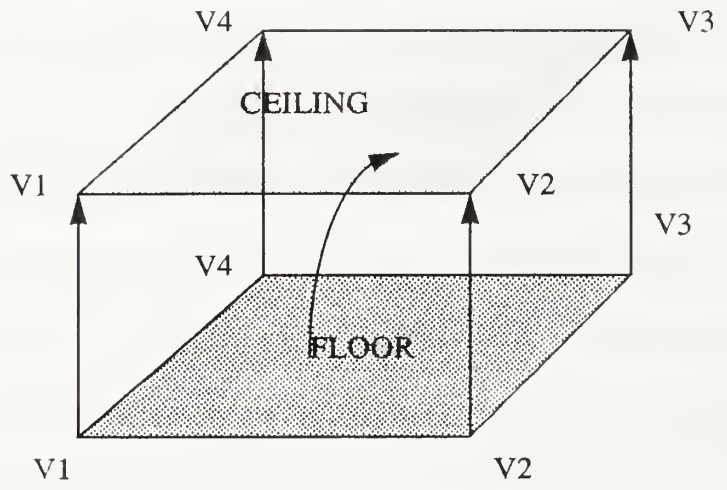


Figure 3.2a

2d+ model
square obstacle representation

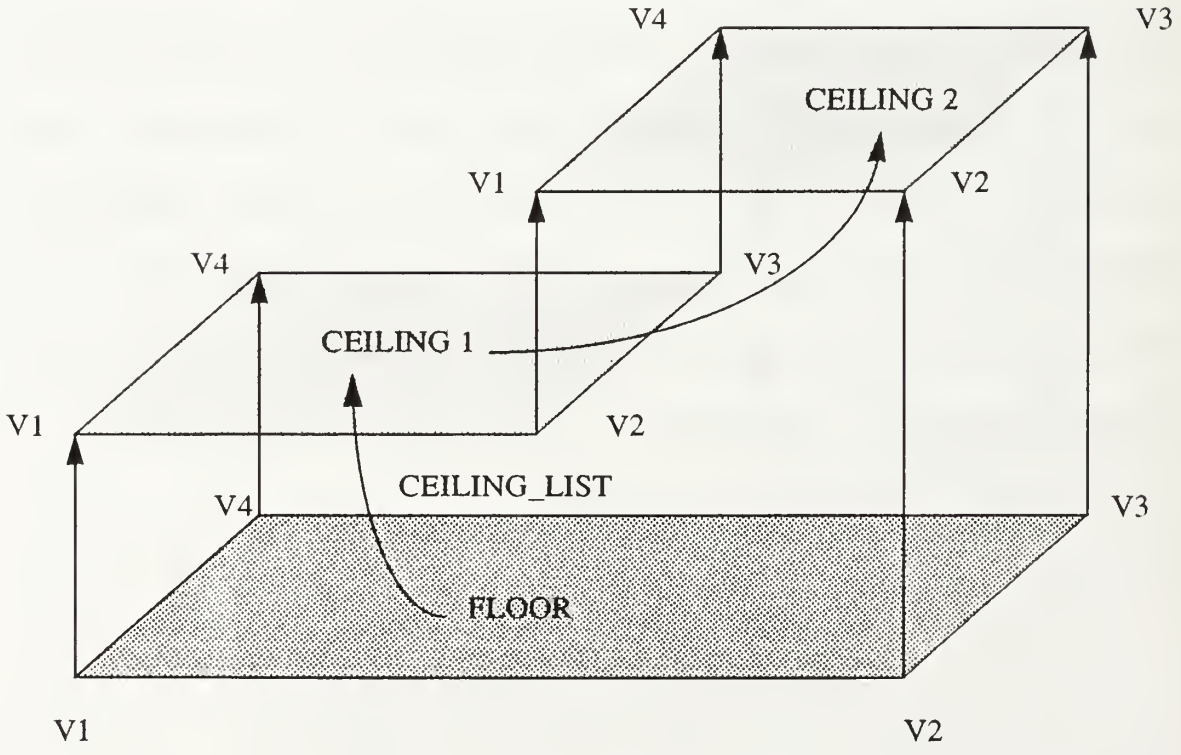


Figure 3.2b

2d+ model
complex obstacle representation

1. The Interface

This 2D+D model will require significant pointer and storage manipulation, but each vertex in a world need only be stored at most once (and the z value stored only once per polygon). We discovered during construction of the original surface model that an interactive interface must address many areas. In particular the interface must provide facilities for easy model creation and addition as well as component structure modifications and deletions. Storage to and reconstitution from disk files was also necessary to save model information which had been entered.

With the added complexity of the 2D+D model's numerous pointers, it became apparent that construction of a useful, interactive interface would be very time consuming and the final product cumbersome to use. For this reason, a simplified interface was decided upon which still affords users the necessary functionality.

A core of functions for construction of a 2D+D world has been provided in the tool file, **2d+d.h** (see A-3). To create a model, a generation file of declarations and function calls is made which uses these tools to make the various structures and assigns them the correct information for the desired world. This generation file (or function) can then be compiled and run with the resulting world structure sent to whatever routines are desired (graphic display, path planner, pattern matching, etc.). This method has the dual advantage of utilizing a text editor for world modifications and negating the need to store and recover information to binary disk file after use. Functions are provided as tools for constructing a model, textual display of the world and memory allocation/deallocation.

2. Two Dimensional Path Planning

Polygons and instances linked into their parent polyhedron structures will be sorted by their z value. This will allow easy determination of surfaces that will obstruct YAMABICO's movements by simply considering all polygons appearing along the height of the robot (typically 0 to 42 inches) as input to the path planning algorithms (after proper filtering merges overlapping polygons). Processing to determine the 2D projection of the model will take little time and therefore should readily support future calculations of path corrections in transit.

Routines for Yamabico's 2D path planning are currently lisp based and assume an input world of polygons from a text file. Future versions of these routines will be implemented in C, and work directly on the two dimensional structures residing in the 2D+D model. Until these modifications are implemented, a process may need to be written which generates the appropriate text file for input to the current path planner.

E. SUPPORT FUNCTIONS

1. Overview

As mentioned above, there are several functions which must be provided to adequately implement the model we have decided upon. In general these fall into three groups: model construction, visibility checking and standard graphic support.

In this chapter we will briefly review the set of functions which directly supports construction of the 2D+D model. These functions reside in the file **2d+d.h** and provide

the facilities for memory allocation/deallocation, pointer manipulation and data insertion which are used to build the dynamic memory representation of our model.

Chapter IV will discuss a visibility checking algorithm and the functions in file **visibility.h** used to implement it. This algorithm is used to determine which edges in the model can be seen from a given position (x,y,z). Visibility is computed without regard to camera orientation and view angle, the standard graphic functions will be relied upon to filter out edges affected by these factors.

Chapter V describes the graphic functions from the file **graphics.c**. These are required to extract a set of lines from the 2D+D model for use in pattern matching. The generalized forms of these functions can be found in any standard graphics support library, but we have tailored them to work directly on our model.

2. Model Construction

This section provides a brief, general description of the functions which provide direct support to the 2D+D model. Our intention is not to provide full directions for the usage of these function as that will be addressed in the users manual, chapter 6. These functions fall into five general categories: memory allocation, memory deallocation, model construction, data display and data location. We will briefly discuss each of these categories and why they are needed.

Memory allocation routines (A-4) have been written to allow for easy creation of the objects(C structures) used in the model. These functions each return a pointer to a structure which has been created using the 'malloc' command from the C language. All

components of the structure created are initialized to appropriate values (generally NULL for pointers, blank spaces for characters and zero for other types).

The functions have been named to reflect the structures they create (i.e. *create_world* returns a pointer to a new WORLD structure). These creation routines are used throughout the files **2d+d.h** and **5th.h** for the creation of polyhedron, polygon, vertex, instance, and world structures used in construction of the 2D+D model.

It is, of course, necessary to provide memory deallocation routines to release memory, which is no longer needed, back to the control of the resident operating system. The basic C command 'free' is appropriate for releasing the memory held by individual structures. Three routines have been written to deallocate the memory used in linked lists, these are: *free_pg*, *free_ph* and *free_world* (A-6). The first two functions are used to deallocate lists of polygons and polyhedra respectively. The *free_world* function makes calls to the other functions and is used to free all of the memory used to store a world.

Six model construction functions (A-9) provide facilities for building the 2D+D representation of a world. The first four of these functions allow creation and addition of vertex, polygon, polyhedron and instance structures to a world. Again, these routines are labeled to match their functionality (i.e. *add_vertex, add_pg, add_ph* and *add_instance*). Each of these four functions accepts information needed to create the object being added and the label of the parent structure to which that object will be added. The parent structure of a vertex is a polygon, of a polygon or an instance is a polyhedron and of a polyhedron is a world structure. When the above functions are used to add vertices to a particular polygon, the vertices are linked in the order they are

added. Likewise, polyhedrons are added to the world in the order they are created. However, when polygons and instances are added to a polyhedron they are ordered by ascending z values. Return values from these functions are pointers to the newly created (and added) structures.

In the previously discussed functions, facilities for building polygons were presented. Since all polygons in a 2D+D model are horizontal, it is necessary to provide methods for assigning the pointers which represent vertical edges. We also need a way to identify which ceilings polygons top the floors of each object. The remaining two functions, *add_edge* and *add_ceiling*, allow this information to be added to a world via simple pointer manipulation. Input parameters to *add_edge* identify two vertices. The vertical edge pointer from the first of these is set to the address of the second, representing a vertical edge in the world. *Add_ceiling* accepts pointers to two polygons as input, and the first polygon is added to the second polygon's list of ceiling.

The data display functions *display_pg* and *display_ph* print the information associated with polygons and polyhedra to the standard input output (stdio) device. The *display_world* function uses both of these to display a textual listing of an entire world to the stdio.

The data location category currently contains only one function, *find_ph*. The input parameters are a string (array of characters) and a 2D+D world structure pointer. The string is used to search through the world for a polyhedron with a matching label. Once it is found the *display_ph* function is called to list the various components of that polyhedron.

IV. VISIBILITY CHECKING ALGORITHM

A. PURPOSE

When a view is being extracted from our model, it is desirable for that view to accurately reflect what a person (or camera) would see if they were to stand at the same point in the physical world. To satisfy this requirement, we cannot allow objects to be transparent. Accomplishing this in the 2D+D model is rather tricky, since polygons representing the walls of objects are not explicitly represented. We first developed a 2D sweep algorithm which quickly determines the set of visible lines (or edges) in a two dimensional world. This algorithm was then expanded to determine the visible lines in a three dimensional space. We will detail out the 2D algorithm and then show the modifications implemented to produce the 3D version. Although the 2D version is quite fast and accurate, we will discuss some inherent problems with the 3D version which limit the output to a close approximation of the set of visible lines. In both cases, the set of visible lines represents an unrestricted 360 degree field of vision.

B. 2D SWEEP ALGORITHM

We will assume standard polygon representation is being used to store vertices in the x-y plane. In this representation vertices of a normal (obstacle) polygon will be chained together in counter clockwise (ccw) order while those of an inverse (enclosure) polygon will be in clockwise (cw) order. In either case the function prev(V) represents the vertex which preceded the vertex V in a polygon. Additionally, let the reference point (rp) be the location of the camera in the plane and the function intersection(E,A) represent the

point of intersection of the edge E with a ray drawn from the rp along angle A. Two lists must also be maintained: the sweep list (a list of endpoints ordered by theta) and the considered list (a list of lines ordered by distance to the rp).

The 2D sweep algorithm follows:

```
1  for each  vertex in a set of polygons loop
2      -calculate the angle, theta, from the rp to vertex
3       normalized to fall in 0-360 degrees
4      -insert in sweep list ordered by ascending values
5  for each  vertex, V, on the sweep list loop
6      if circuit(rp,V,prev(V)) is ccw and V.theta>prev(V).theta then 7          -place
edge (V,prev(V)) on the considered list since it
8       straddles zero degrees
9      -set first point of edge to intersection (edge,0)
10     for each item, V, on the sweep list loop
  //remove all edges ending at V prior to adding the edge starting at V
11         for each edge, E, on the considered list
12             if V is the second point of E then
13                 if  E  is  the  first  item  on  the  considered  list  then
14                     -accept it as visible
15                     -change  the  first  point  of  the  next  item,  E1
16                      to the intersection(E1,V's theta)
17                 -remove E from the considered list
18             -recalculate the distance from rp to E along V's theta
19         if the circuit from rp to V to prev(V) is ccw then
20             if  distance  to  V  is  less  than  distance  to  1st  edge  of
21             considered list, E1 then
22                 -accept E1 rp intersection with V's theta
23                 -change 1st point of E1 to
24                  intersection(E1,V's theta)
25             -insert edge (V,prev(V)) in the considered list
```

Figure 4.1 demonstrates this execution of the algorithm for a simple 2D world. For the purposes of this algorithm, an angle of zero degrees is defined as a line from the rp which runs out in the positive X direction and parallel to the X axis. After all the vertices have been sorted by their theta values and added to the sweep list, processing of edges

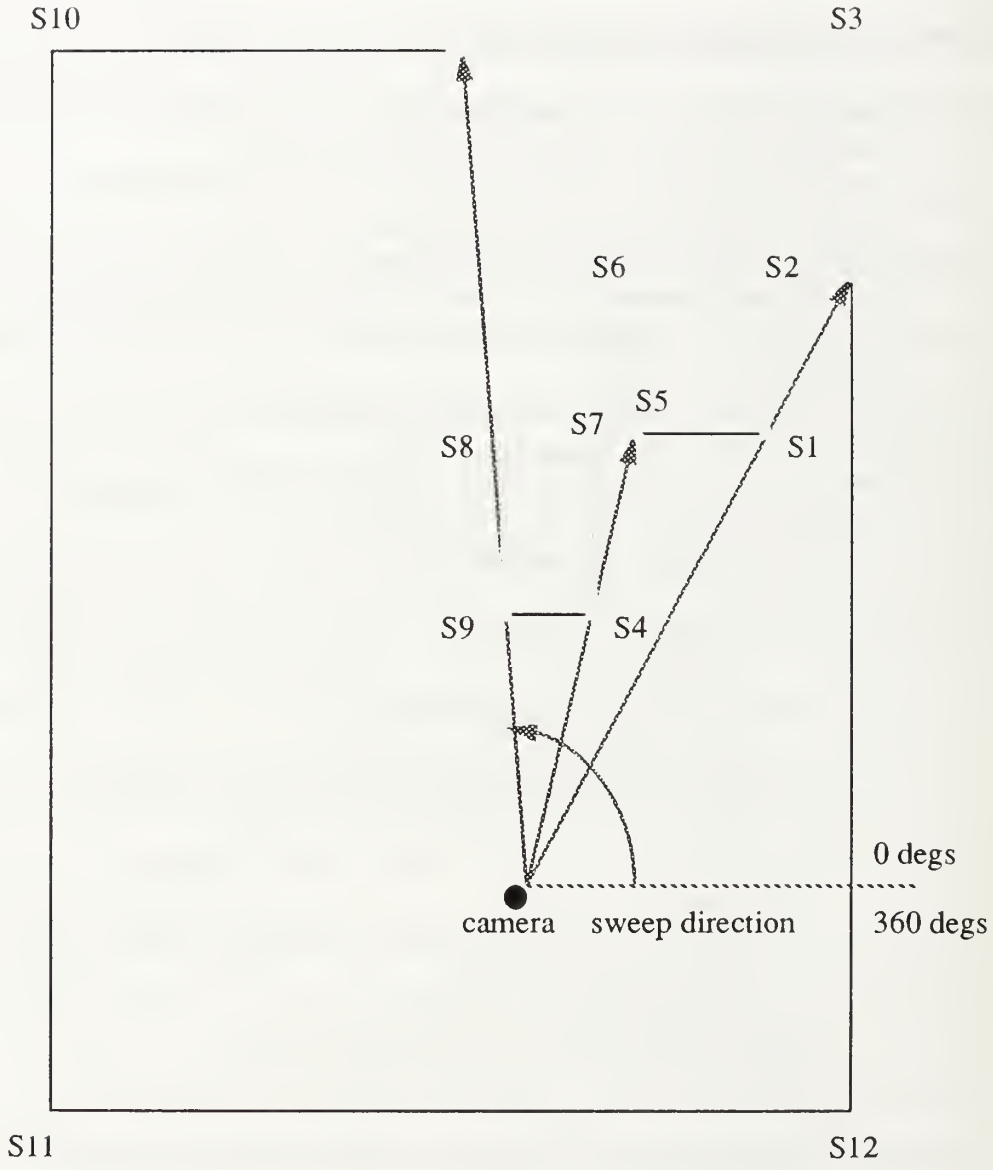
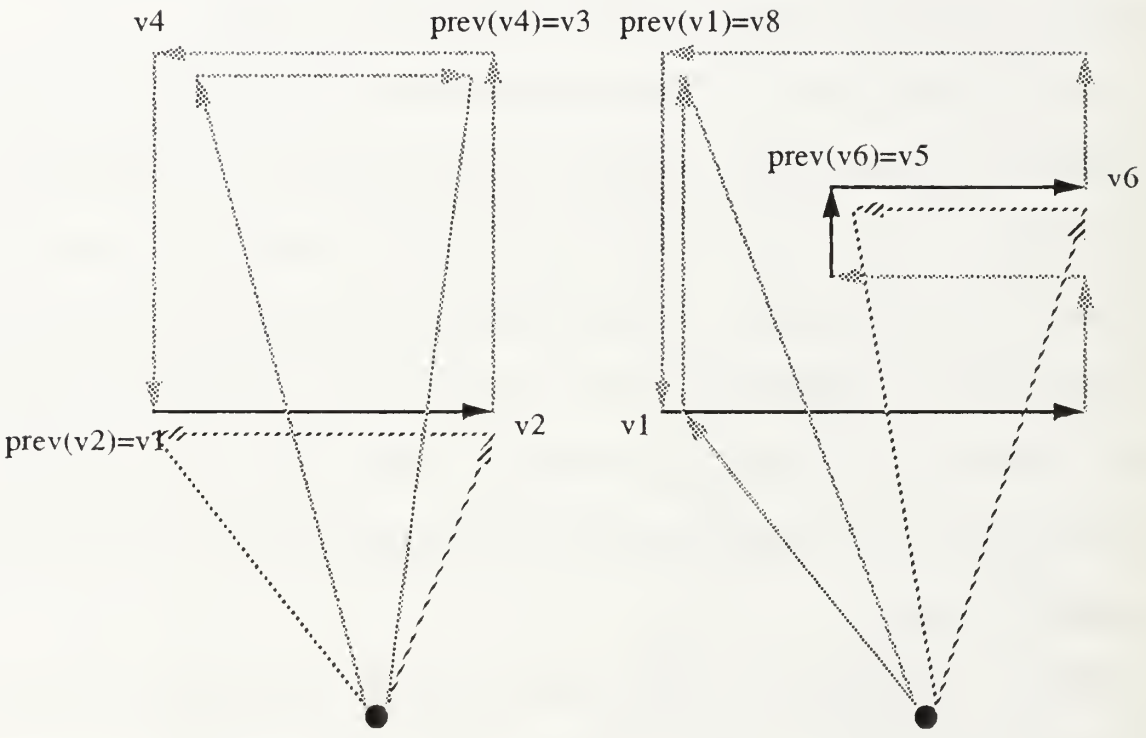


Figure 4.1
Sweep method of
Visibility testing

begins. The direction of an edge is from V to prev(V). If this direction is ccw with relation to the rp, there is a possibility the edge is visible so it is added to the considered list. If the direction of the edge is cw it is occluded from view by some other edge of the polygon it is part of (see Figure 4.2). To ensure the effects of those edges straddling zero degrees are not lost, they are added to the considered list (lines 5-8) prior to processing the entire sweep list. Reassignment of each straddler's first point (line 9) keeps us from blindly accepting the portion lying before zero degrees.

In the main loop (lines 10-25) each vertex of the sweep list is processed. If an edge is under consideration which ends with the current vertex, it is removed from consideration. In a 2D world there can only be one edge visible at any given point around the sweep. Therefore if the edge being removed happened to be first on the considered list, then it was closest to the rp and is accepted as visible (line 14). When this is the case the next edge on the list becomes visible, and the first point must be adjusted (line 15) so the portion occluded by the edge being removed will not later be accepted as visible.

Once all lines ending at the current vertex are removed, the edge from the current vertex to its predecessor is inspected. If the edge is ccw then it is added to the considered list. If this new edge is the closest to rp, the former head of the considered list is accepted as visible up to the current vertex's theta and its first point adjusted accordingly (lines 20-24). Notice that the main loop will remove straddling edges with the artificial first vertex at zero degrees, and then adds them again in the normal sequence. This



(a) convex case

(b) concave case

Figure 4.2
ccw visibility test on
polygons

allows non-straddling edges to still occlude the first portion of straddling edges when they are added to the considered list a second time during the main processing loop (line 19).

The algorithm is complete when the last straddling edge has been removed from the list and each vertex on the sweep list has been processed at least once. Notice that only one edge is visible at any given time and the endpoint reassignments (lines 15 and 24) ensure that only the visible portions of an edge are accepted.

C. 3D SWEEP ALGORITHM

When sweeping for visibility in a 3D world distance to edges is no longer sufficient criteria to determine visibility. Since occlusion of more distant edges may be total, partial or not occur along the Z axis, there can be any number of edges visible at a given point around the 3D sweep. We also have the added need to accept vertical lines which may reside at an edge's endpoints. The basis of the 3D algorithm is the same as that for the 2D sweep, but we need to add some information which denotes an edge's presents along the z axis. This information effectively defines each edge as a vertical wall of some height greater than or equal to zero.

Specifically, each sweep list item must indicate if a vertical line begins at that vertex or not. Since we will be interested in determining which edges are occluded from the view of a single point, rp, in 3D, simply comparing z information of edges and these vertical lines is insufficient. For this reason, when a sweep link is formed the angle of elevation from the rp to the vertex is calculated and stored in the variable MIN_Z. Likewise, if a vertical line is present at the vertex, the variable MAX_Z stores the angle

of elevation to the upper vertex of that line. A vertex which has MIN_Z equal to MAX_Z will by definition have no vertical line associated with it.

As with each sweep list item, each item of the considered list will also contain a C_MIN_Z and C_MAX_Z angle. In the sweep list we are concerned with z information mainly to represent vertical lines. In the considered list we need z information which accurately reflects the extent of occlusion an edge can inflict (from floor to ceiling) on edges behind it. Figure 4.3 demonstrates a case where z information for the considered list edge must be different than that of either endpoint. The function *find_ceiling_z*(E) will be used to provide the height of the ceiling which tops a particular edge E.

As mentioned earlier, many edges may be visible in 3D at one time. For this reason we have added two flags to each considered item: visible and bottom_visible. The first flag indicates that some part of the edge's plane of influence can still be seen from rp. The second flag indicates that the bottom of the edge is still visible indicating that an output line must be generated if the edge is modified or the sweep passes it with this flag set.

The algorithm we present has been trimmed somewhat to help increase clarity. Several special cases occur within a model which require individual handling. Two such cases arise since the ccw check fails to recognize all edges of a ceiling which occurs below the rp and a floor which occurs above it. In these two cases the sweep list must be artificially manipulated to generate all edges of these polygons as ccw with respect to the rp (so they may be placed on the considered list).

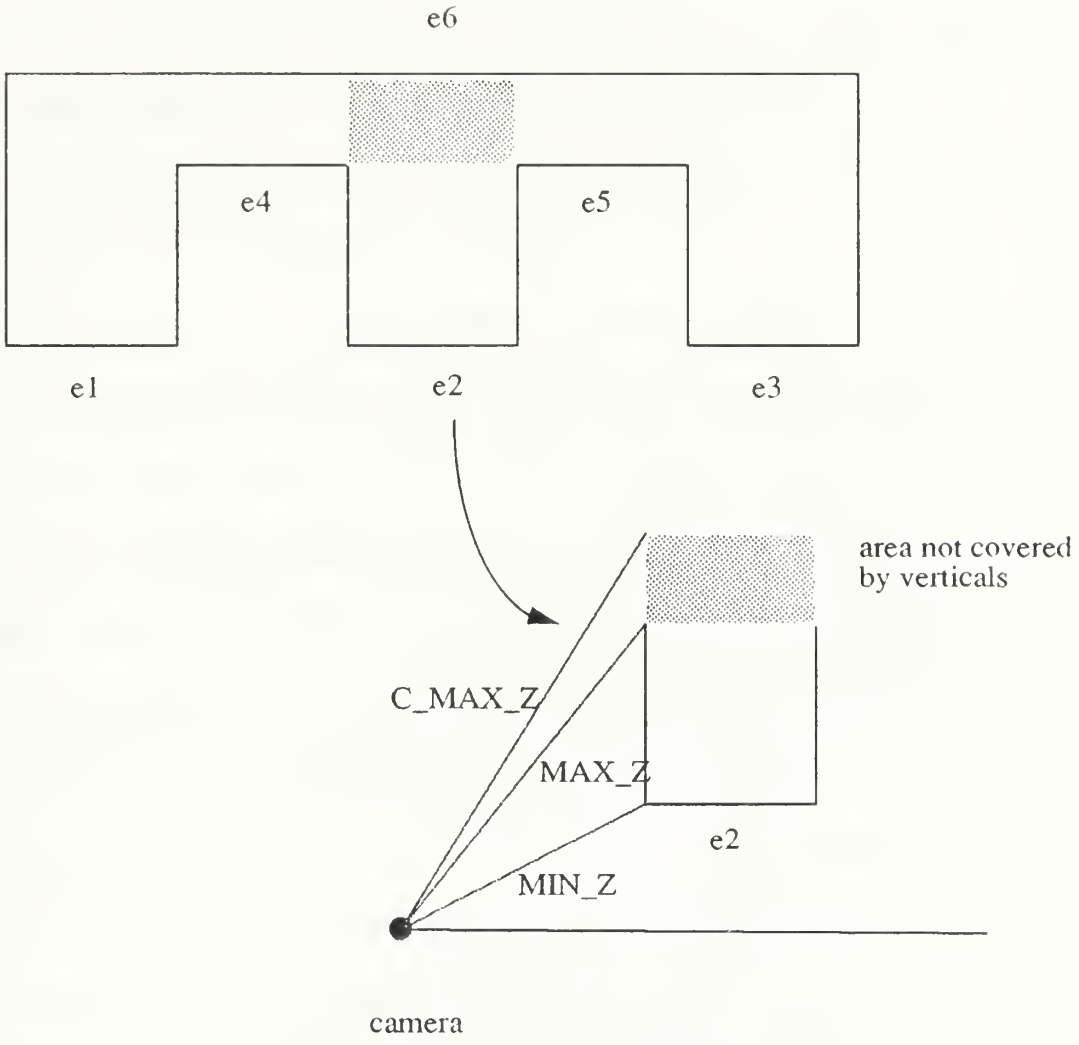


Figure 4.3

A case where the z coverage of the vertical lines originating at an edge's endpointd does not match the coverage of the edge itself.

The 3D sweep algorithm follows:

```
1  for each  vertex in the model
2     -calculate the angle, theta, from the rp to vertex
3      normalized to fall in 0-360 degrees
4     -calculate MIN_Z from rp to V
5     -if V has outgoing vertical edge then
6        -calculate MAX_Z to top of vertical
7      else
8        MAX_Z=MIN_Z
9     -insert in sweep list ordered by ascending values
10    for each  vertex, V, on the sweep list loop
11       if circuit(rp,V,prev(V)) is ccw and V.theta>prev(V).theta then 12 -
place edge (V,prev(V)) on the considered list since it
13        straddles zero degrees
14       -let C_MIN_Z = V.MIN_Z
15       -calculate C_MAX_Z based on find_ceiling_z(edge(V,prev(V)))
16        so angle falls between -90.0 and 90.0
17    for each item, V, on the sweep list loop
18       for each edge (E) on the considered list
19          if V is the second point of E then
20             if E's 2nd point has vertical edge then
21                -calculate visibility of vertical edge and
22                 accept part of vertical line seen
23             if E's visible=1 then
24                -accept it as visible
25             -remove E from the considered list
26             for each edge on considered list loop
27                -recalculate C_Z_MIN and C_Z_MAX
28             -calculate_visibility(considered list,V.theta)
29          -recalculate the distance from rp to E along V's theta
30       if the circuit from rp to V to prev(V) is ccw then
31             if V has vertical edge then
32                -calculate visibility of vertical edge and
33                 accept part of vertical line seen
34          -insert edge (V,prev(V)) in the considered list
35          -for each edge on considered list
36             -recalculate C_Z_MIN and C_Z_MAX based on V's
37              theta (since perspective changes angles)
38          -calculate_visibility(considered list,V.theta)
```

Execution of the 3D version closely parallels that of the 2D version, with some notable enhancements. Since vertical edges can be associated with sweep list vertices, we calculate the visibility of the first endpoint's vert edge when a new considered list item is added (lines 20-22) and the second endpoint's vert edge when that considered item is removed (lines 31-33). This calculation simply consists of looping through all the edges appearing in front of the current edge on the considered list. For each of these edges the MIN_Z and MAX_Z values of the vertical edge under consideration are adjusted to reflect any occlusions. When the loop is complete, if MIN_Z is less than MAX_Z the visible portion of the vertical edge is accepted.

Another essential, and expensive, addition to the 3D algorithm is the recalculation of the elevation angles on each considered list item (lines 27,36). Since the occlusion of the z axis by an edge is represented by limiting angles (C_MIN_Z,C_MAX_Z), perspective must be considered. We cannot simply calculate the z coverage of an edge when it is added and assume it to be constant. The perspective changes with the distance to an edge, so as the sweep progresses, the coverage of edges pointing towards rp increase while those pointing away decrease. These calculations are performed after the addition or removal of a considered edge and are followed by a new calculation of all edges visibility.

The calculate_visibility function (lines 28,38) is the 'work horse' of the 3D sweep which assigns visibility and adjusts Z coverage in response to occlusion. This function is executed each time a change is made to the considered list. It scans through the list and for each edge compares the Z coverage information to the edges behind it. The

visibility flags, C_MIN_Z and C_MAX_Z are adjusted based on if and how each edge is occluded. An additional variable associated with each edge on the considered list, MIN_SWEEP is introduced in this function. MIN_SWEEP keeps track of the starting sweep angle for which the visibility information is valid. When part of an edge is accepted as visible or its status changes, the MIN_SWEEP must be updated. A basic description of the function follows:

```
assumptions: all visibility flags set
        C_MIN_Z and C_MAX_Z set to total coverage
                (initially no occlusion)

input: THETA = current sweep angle
       considered list

   for each edge, E on the considered list
      for each edge E1 farther down the considered list than E
         -determine how E occludes E1
         -case type of occlusion:
            entirely occluded:
               visible=bottom_visible=0
            bottom occluded:
               bottom_visible=0
               E1.C_MIN_Z=E.C_MAX_Z
            top_occluded:
               E1.C_MAX_Z=E.C_MIN_Z
   for each edge, E on the considered list
      if above loop changed E and E.bottom_visible=1 prior to
                        the loop then
         -accept E from intersection(E,MIN_SWEEP) to
               intersection(E,THETA) as visible line
         -E.MIN_SWEEP=THETA
```

D. PROBLEMS WITH THE 3D SWEEP ALGORITHM

Two major concerns arise when reviewing the usefulness of our 3D sweep algorithm: can it support real time image processing and is the set of output lines correct.

Although the final goal of this work is to support a real time vision system, a hard and fast definition of 'real time' is difficult to express. This is especially true in the case of providing support to pattern matching facilities, which themselves may take more than 30 seconds to execute. In general, we would consider programs with a total run time under two seconds to be classified as real time applications. Unfortunately, as the number of polygons comprising our model increase to the number required to represent the target environment (fifth floor hallway), we find run time increasing to approximately eight seconds. In a hard real time system such a lag time would almost assuredly be unacceptable, but if we take into consideration the amount of processing time likely to be required by the entire vision system such performance may be tolerable. For this reason we will classify the visibility algorithm as having 'near' real time performance.

The reason for this poor behavior can be attributed to the need to account for perspective. Recalculating the z axis coverage and the visibility for the entire considered list whenever an edge enters or leaves the list and at each increment of the sweep angle becomes quite expensive. Even so, we still see that some inaccuracies in output can occur since z coverage should ideally be updated continuously.

When reviewing the correctness of the set of visible lines generated we find several small discrepancies. One of these is the above stated problem of not being able to continuously update z coverage. Associated with this is the fact that, to save time, our algorithm does not consider the possibility that the relative positions of edges on the considered list may change. Figure 4.4 illustrates how the distance to edges may call for a resorting of the considered list as the sweep progresses. The benefits of resorting the

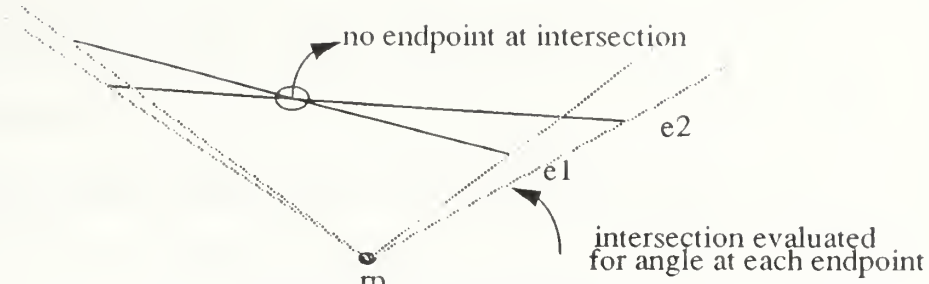
list at each sweep increment is overshadowed by the significant increase in processing time which results.

The above section discussed the 'artificial manipulation' used to ensure floors occurring above the rp and ceilings occurring below have all edges ccw. Those ceiling edges which are forced to be ccw in general have no outgoing vertical edges and therefore no associated z coverage and are therefore rendered transparent (see Figure 4.5).

The last two items problems have been knowingly designed into the algorithm and are not apparent from the previous description. When vertices from enclosure ceilings are being entered on the sweep list, they are automatically given a MAX_Z of 90 degrees. This is the only simple way to prevent such ceilings from being totally transparent (although edges directly over the ceiling can still be seen).

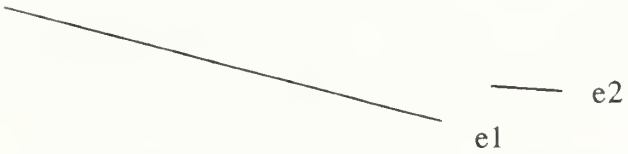
Lastly, to ease the complexity of our data structures, edges are not considered to be occluded across the middle along their entire length (Figure 4.6). When this case does occur, we allow the occluding object to cast a footprint by assuming the occlusion is across the bottom along the entire length. To model the true situation, we require a representation which can track more than one non-contiguous area of coverage along the z axis. This would require a list of (MIN_Z,MAX_Z) pairs with splitting and merging functions to replace the single variables our algorithm uses.

Chapter VI compares usage of the full 3D sweep algorithm with some quicker, but less accurate, implementations of the sweep method. Timing results and correctness of output are both discussed.



(a) initial viewing situation

(b) correct set of visible lines



(c) actual output of 2d sweep algorithm

Figure 4.4
Problem due to perspective with sweep algorithm

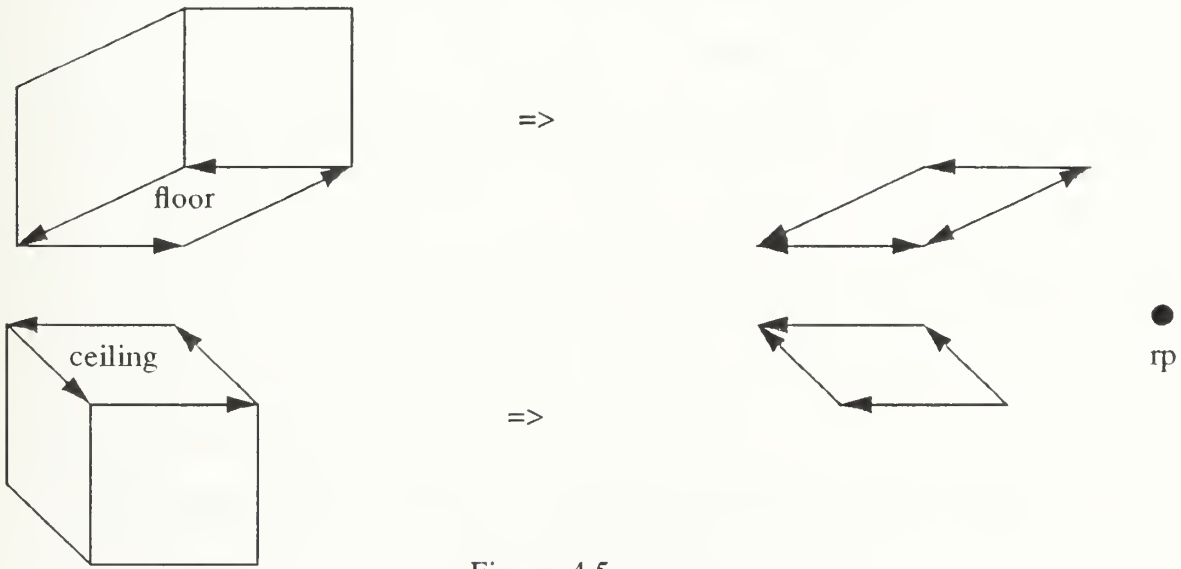


Figure 4.5
Artificial manipulation to make
all edges ccw.

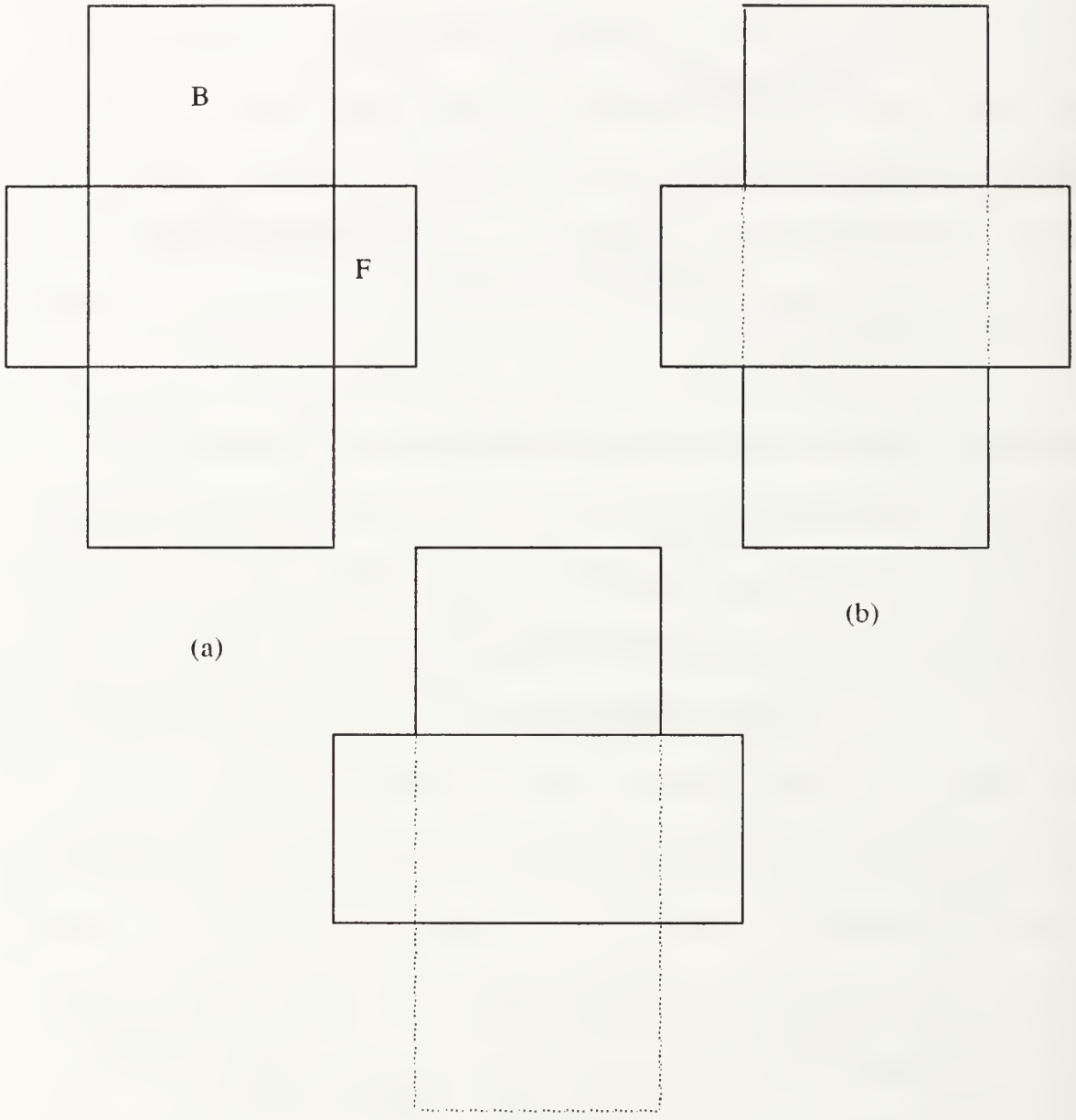


Figure 4.6

a. original case b. desired occlusion c. approximated result from visibility sweep algorithm

V. STANDARD GRAPHIC SUPPORT

A. OVERVIEW OF OUR APPLICATION NEEDS

Although the personal iris system on which this work currently resides does provide an extensive graphic support library, our vision system is being designed to reside on board Yamabico. Since Yamabico's memory (both primary and secondary) is somewhat limited and source code for the iris library is not available, it was necessary to write our own graphic support functions.

After the visibility checking algorithm has been run on a 2D+D world, we are left with the set of all lines which are visible from a specific point in the model. The reader should recall that output from visibility checking does not take the orientation of the observer (camera) into consideration but rather, provides a complete set of theoretically visible lines based upon omnidirectional sensors. The graphics support routines will determine which of these lines fall within the camera's field of vision and transform them into a final format which can be used in pattern matching and graphic display. Since our vision system will exploit a single camera as it's sole sensor, the processed image it provides will be composed of 2D lines. Likewise, a standard display terminal can only support drawing lines specified in it's 2D screen coordinate system. For these reasons, the appropriate final format for our view from the model is a set of 2D lines. As with the 3D lines provided by the visibility algorithm, we choose to use an endpoint representation to specify these 2D lines.

[Ref. 13] thoroughly describes the mathematics and principles of computer graphics. It was the primary reference used to design the support functions found in the file

graphics.h. When a view from the model world is needed, the *get_view* function is called. This function requires a pointer to the 2D+D world and the camera's 3D position (PRP), orientation (DOP) and field of view in degrees (view_angle) as input. The world and PRP are sent to the visibility checking algorithm and the returned list of lines is worked with from then on.

Since we are simulating what is seen by a single camera, we need to extract a view from our model which is based on a single point perspective projection. In a perspective projection, line size is scaled to the inverse of the distance from the camera. This allows distant objects to appear smaller than closer ones of the same physical dimensions. A parallel projection does not perform this scaling and is therefore not suitable to our application.

B. GENERAL PERSPECTIVE PROJECTION

There is a pipeline of several steps which is used to produce a perspective projection from a model. The steps are:

1. Define the View Volume

Each candidate line, in a model, must be inspected to determine if it falls within the observer's field of vision. This field of vision is defined by a semi-infinite pyramid originating at the PRP and extending along the DOP (Figure 5.1). All lines within this volume will be seen by the observer. The infinite length of this pyramid is difficult to work with, so we define a near clipping plane and far clipping plane. These are defined relative to the observer and form the truncated pyramid of Figure 5.2. The height and

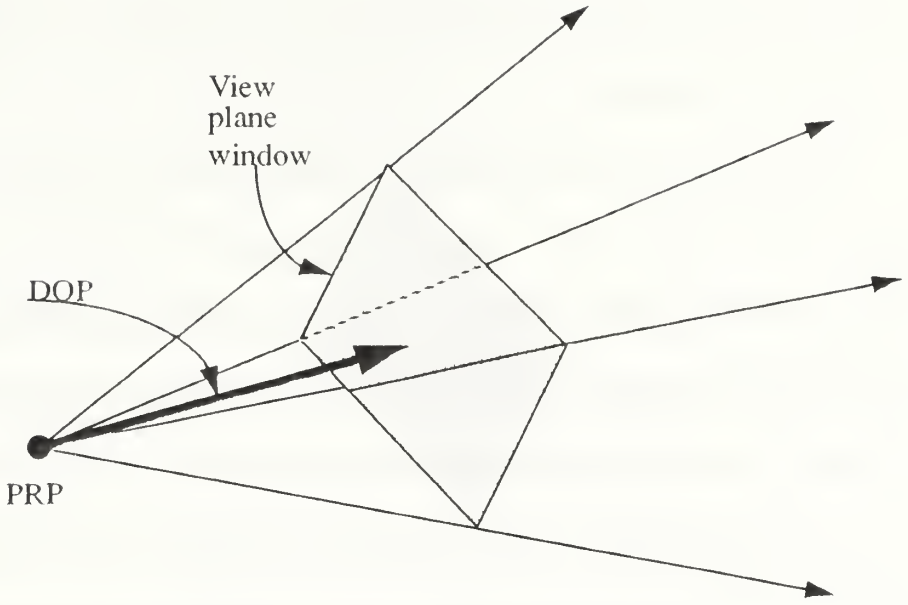
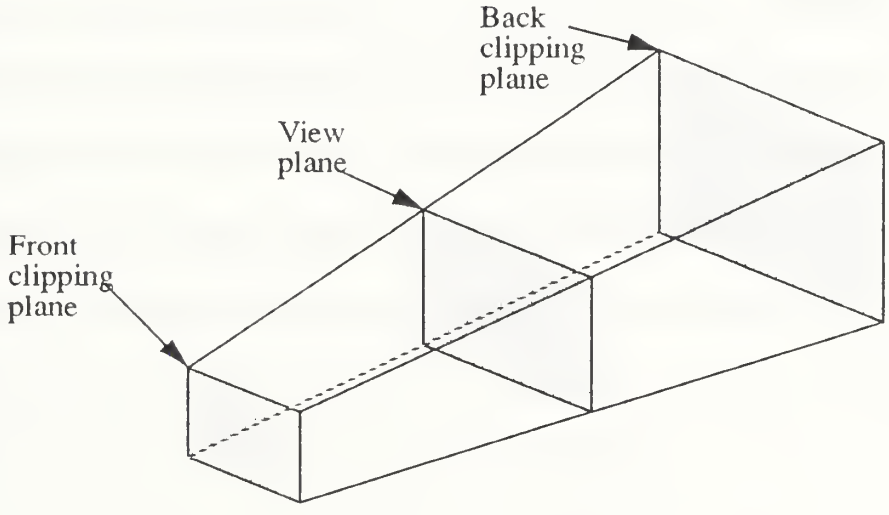


Figure 5.1
Semi-infinite pyramid defining
view volume for perspective projection



PRP

Figure 5.2
Truncated view volume.

width of the pyramid at the near and far clipping planes is determined by the width of the observers view_angle, or the defined size and location of the window (see section b).

2. Select a 2D Window.

Somewhere along the length (along the DOP) of the pyramid a projection window must be placed. This window is parallel to the base of the pyramid and is the surface onto which all accepted lines will be projected. Lines falling within the pyramid on the far side of the window from the observer, will be scaled down in size when mapped to the window, while those on the near side will be scaled.

If the view_angle is not used to define the slope of the view volume's sides, a window can fully define the volume by giving its height, width, and distance down the DOP from the observer.

3. Determine the Normalizing Transformation

Although it is simple to determine if a line falls between the far and near clipping planes, it is difficult to calculate what portion of a line (if any) falls within the other four sloped planes which make up the sides of the pyramid. To ease these calculations, the pyramid is manipulated to form a unit pyramid which defines a canonical view volume. The new pyramid's surfaces or clipping planes are represented by the following equations:

right: $x = -z$ (5.1)

left: $x = z$ (5.2)

bottom: $y = z$ (5.3)

top: $y = -z$ (5.4)

front: $z = z_{min}$ (5.5)

back: $z = -1$ (5.6)

The general form of this transformation, N_{per} is:

$$\mathbf{N_{per} = S_{per} * SH_{par} * T(-PRP) * R * T(-VRP)} \quad (5.7)$$

where:

T(-VRP) Translate the view reference point (VRP) to the origin. For each 3D point, P, which is being normalized, add the negative of the corresponding VRP coordinate to each coordinate value of P. The VRP is the origin of the view coordinate system. Since window limits are referenced from this point, it is a good idea to choose a VRP position which readily supports window reference (i.e., center of the window or a window corner).

R Rotate the view reference coordinate (VRC) system so it is aligned with the (x,y,z) system. The VRC system has three components (u,v,n). Initially (Figure 5.3a) v is vertically aligned with the window, u runs parallel to the lower edge of the window and n is normal to the window surface. Proper alignment is achieved by rotation about the x, y and z axis. Rotation takes advantage of the trigonometric sine and cosine functions of the rotation angle (RA). These functions are applied to the original coordinates of a point, (x,y,z) to produce the new point, (x1,y1,z1). As an example, equations to determine rotation about the z axis are shown:

$$x1 = x * \cos(RA) - y * \sin(RA) \quad (5.8)$$

$$y1 = x * \sin(RA) + y * \cos(RA) \quad (5.9)$$

$$z1 = z \quad (5.10)$$

T(-PRP) Translate the PRP to the origin. The PRP is also known as the *center of projection* and refers to the position of the observer or camera.

$\mathbf{SH_{per}}$ Shear the view volume along the z axis so the DOP is parallel to the z axis. Multiplication by a shearing matrix will augment the x and y terms to accomplish this. We will not go into detail on how to derive this matrix, since shearing is not needed in our application. The interested reader is referred to Reference 13 page 264.

$\mathbf{S_{per}}$ Scale the view volume into the canonical perspective-projection view volume. We must determine a scaling factor for each coordinate system axis, which is multiplied by the corresponding (x,y or z) component of the point to be scaled. Here the goal is to map the back clipping plane so its new location is at z=-1. The apex of the view volume will map to z=0, leaving the front plane (located on z_{min}) at its relative position between the two. Equations 5.1-5.6 must hold true after this scaling. To guarantee this, all z components are scaled by $\mathbf{-1/(vrp_z'+B)}$. The denominator of this term is simply the position of the back clipping plane after it has been processed through the previous normalization steps. Remember that we are targeting clipping plane equations with unit slopes, x=z and y=z. The z scaling factor must also be applied to x and y to ensure uniformity, but we first must scale to produce these unit slopes. This is accomplished by scaling the window half-height and half-width to vrp_z' (since this is the transformed z position at which the window now resides). Therefore the appropriate scaling factors for x and y are:

$$(2*vrp_z')/((vrp_z'+B)*(window\ width)) \qquad (5.11)$$

$(2*vrp_z')/((vrp_z'+B)*(window\ height))$ (5.12)

respectively.

4. Apply the Normalizing Transformation to All Lines

Each of the lines from the model are transformed and those that fall within the normalized canonical view volume will be seen.

The interested reader will note that [Ref. 13] expresses all of these manipulations through use of matrix operations. To save the expense of writing a math package for matrices our application uses series of linear equations to emulate matrix use. Although these equations are basically equivalent, they may appear somewhat different since the problem we are solving is a subset of the general case.

5. Clip Normalized Lines Against Canonical View Volume

Each endpoint of a line will be assigned a six bit clipping code. The coordinates of the endpoint are compared to the equations of the canonical view volume's planes. The meaning of each set bit follows [Ref. 13]:

bit 1 - point above view volume	$y > -z$
bit 2 - point below view volume	$y < z$
bit 3 - point to right of view volume	$x > -z$
bit 4 - point to left of view volume	$x < z$
bit 5 - point behind view volume	$z < -1$
bit 6 - point in front of view volume	$z > z_{min}$

When both endpoints of a line have clipping codes of all zeros, each endpoint falls within the view volume and the line is trivially accepted. Likewise, when a bitwise logical *and* of the endpoint codes does not produce all zeros the line is rejected since it lays totally outside the volume. When neither of these cases is met, only a portion of the line is within the view volume. In this case, the next step is to calculate the intersection(s) with the volume's six clipping planes. This is where the advantage of selecting unit slopes for those planes is realized. An extended 3D version of the Liang-Barsky 2D clipping algorithm is employed to find the intersections [Ref. 13 pg. 274]. This algorithm uses the parametric representation of a line:

$$x = x_0 + t(x_1 - x_0) \qquad (5.13)$$

$$y = y_0 + t(y_1 - y_0) \qquad (5.14)$$

$$z = z_0 + t(z_1 - z_0) \qquad (5.15)$$

where t is in the interval (0,1)
and subscripts specify which endpoint a
coordinate refers to.

These equations are set equal in accordance with equations 5.1-5.6 and t is solved for. The two t values which fall in (0,1) define the new endpoints of the partial line to accept. Notice that one t may be 0 or 1 if one endpoint is in the view volume and the other is not.

6. Perform Perspective Projection

Once the 3D lines within the view volume are identified, we need to map them onto the 2D window. This is simply accomplished by dividing each coordinate of the

endpoints by z/d. Where z is the z coordinate of the point and d is the transformed position of the projection plane on the z axis. Notice that this will map the z coordinate of each endpoint to d.

7. Scale Window Coordinates to Device Coordinates

In order for our final set of output lines to be useful, we must map them from the window coordinates to some device coordinates. The lower left corner of our window is (z_{min},z_{min}) by equations 5.11 and 5.12. The width and height of the window are both $2*z_{min}$. If the new device coordinate limits are denoted by XMIN, XMAX, YMIN and YMAX mapping is accomplished by:

$$X = ((\text{x window coordinate}-z_{min})/2*z_{min})*(XMAX-XMIN)+XMIN \quad (5.16)$$

$$Y = ((\text{y window coordinate}-z_{min})/2*z_{min})*(YMAX-YMIN)+YMIN \quad (5.17)$$

C. PERSPECTIVE PROJECTION FOR OUR APPLICATION

1. Define the View Volume

The near and far clipping planes are located relative to the camera position at 1.4 inches and 5000 inches respectively. The near clipping plane is chosen to match the focal length of our camera, and the far plane's distance is greater than the total length of our model world (thus ensuring all lines which should be seen can be).

2. Select a 2D Window.

According to specifications, our video camera has a *ccd* element which is two thirds of an inch square. This element is the camera's physical counterpart to the window on which we need to project the model lines. Using this information along with empirical

testing, we have determined that the focal length of the camera is 1.4 inches. Again, this value corresponds to how far from the camera (along the DOP) the window should be placed.

3. Determine the Normalizing Transformation

Figure 5.3 shows each step of the normalization process. Some of the physical restrictions we place upon our target system simplify the normalization transformation from its general form:

$$\mathbf{N_{per} = S_{per} * SH_{par} * T(-PRP) * R * T(-VRP)}$$

to:

$$\mathbf{N_{per} = S_{per} * T(-PRP) * R_y * T(-VRP)} \qquad (5.18)$$

where:

T(-VRP) Translate the view reference point (VRP) to the origin (Figure 5.3b). We select the lower left corner of our window as this point.

$\mathbf{R_y}$ Rotate the view reference coordinate (VRC) system about the y axis so it is aligned with the (x,y,z) system (Figure 5.3c). Rotational computations have been simplified from a general three coordinate rotation to a single rotation about the y axis. This is due to the fact that Yamabico will only rotate its camera freely in the model's x-y plane[1].

---

[1] It is important to note that most graphics discussions assume a 3d coordinate system where the -z axis goes into the page rather than our model's coordinate system where the z axis denotes height. Therefore all coordinates must be shifted from model to graphic representation prior to performing normalization, and the R_y rotation is actually about the model's z axis, as it should be.

T(-PRP) Translate the PRP to the origin (Figure 5.3d). This will be the position of Yamabico's camera which has gone through the previous two operations.

$\mathbf{S}_{per}$ Scale the view volume into the canonical perspective-projection view volume (Figure 5.3e). This is performed as specified for the general case.

The most noticeable simplification is the omission of $\mathbf{SH}_{per}$. Since Yamabico's camera will be mounted perpendicular to the floor there is no need to shear the view volume. The DOP is always parallel to the z axis.

4. Apply the Normalizing Transformation to All Lines

Each line from the list of visible lines returned from *get_view* is transformed and if it falls within the normalized canonical view volume will be seen in the final output.

5. Clip Normalized Lines Against Canonical View Volume

This step is carried out exactly as for the general case. We have reserved space for the clipping codes within each LINE structure. The function *get_clipping_codes* (A-46) assigns the six bit code to each endpoint, and the functions *clip_line* and *clipt* (A-46) make up our Liang-Barsky 3D clipping algorithm implementation.

6. Perform Perspective Projection

Since we are using the focal length of the camera to position our window, the value of d will be z_{min}. This indicates that our window overlays the near clipping plane.

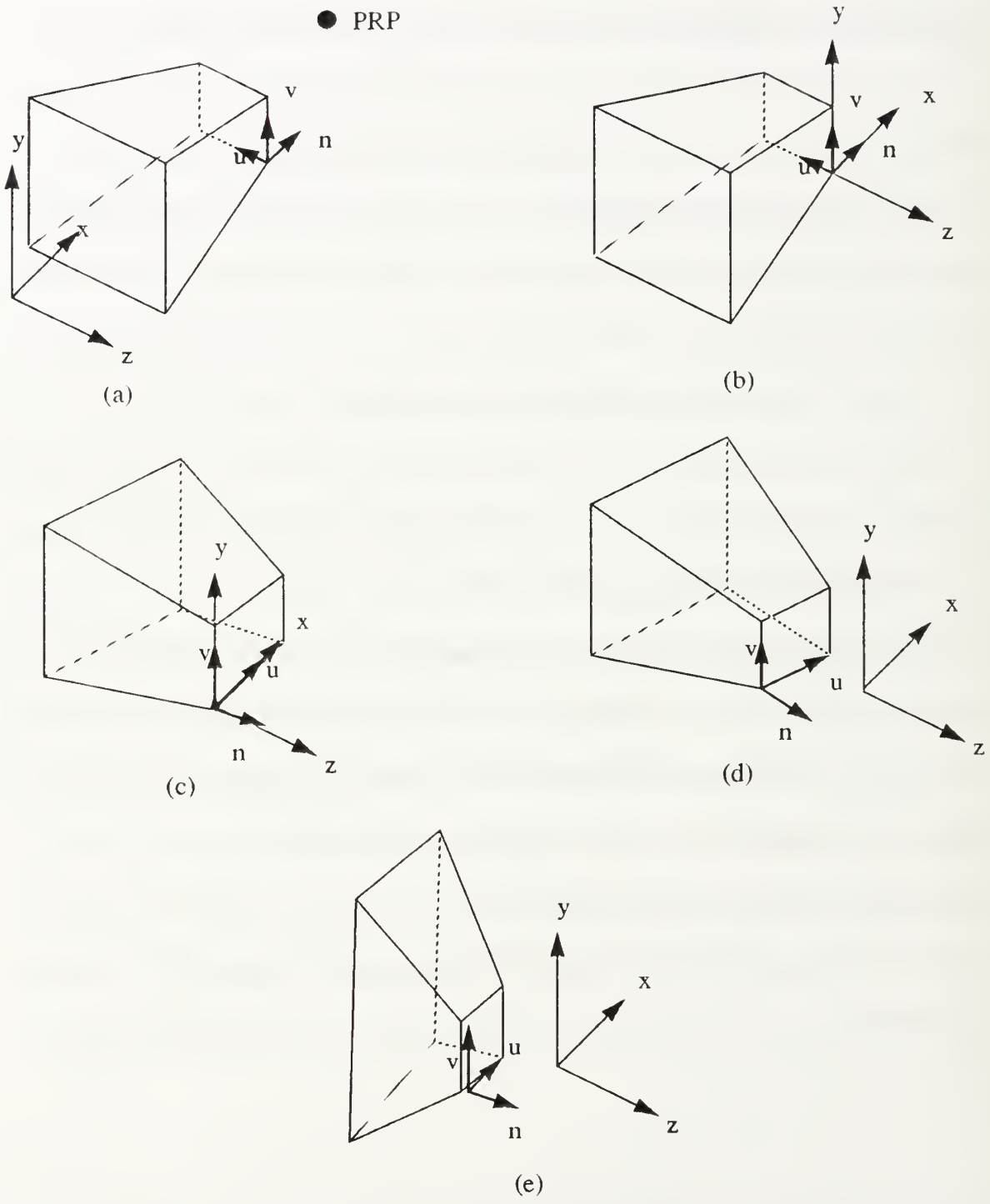


Figure 5.3

(a) Initial viewing situation. (b) VRP translated to origin (T(-VRP)).
(c) Rotated about y axis to align (x,y,x) with (u,v,n) coordinates (Ry).
(d) PRP translated to origin (T(-PRP)). (e) View volume scaled into canonical form (Sper).

7. Scale Window Coordinates to Device Coordinates

The final device for our projected lines will either be a window for pattern matching or a portion of the graphics display screen. In either case, the variables MIN_X, MAX_X, MIN_Y and MAX_Y are defined at the top of the file graphics.h. Changing these variable will allow the *map_to_screen* function to properly scale the final set of lines.

VI. IMPLEMENTATION AND CONCLUSIONS

A. MODEL

1. Appropriateness of 2D+D Model

Our chosen representation has proven quite effective for modelling environments that are orthogonal with respect to the z axis. Although we cannot model curved objects such as door knobs, those objects which contain the major features needed for pattern matching are readily represented.

Path planning has not yet been implemented using the 2D+D model, but the asymmetric quality of the model seems to strongly support such an application. We simply need to treat all horizontal polygons, which have a z value along the robot's height, as objects for the path planner. To constrain the problem complexity, overlapping 2D polygons will need to be merged together.

2. Constraints

The 2D+D model is not useful for outdoors environments nor for ones with many curved surfaces. Additionally, path planning cannot currently work directly from the model. The main problem is that instances of polyhedra classes share storage of vertices in the local coordinate system. As the model stands, a separate representation must be used to store the polygons required for path planning. Note that even if dedicated storage for each instance is allocated, the same problem may occur. This is because overlapping polygons still must be merged, but the underlying model must not be altered.

Another constraint is placed on model construction. When a polyhedron is added we only allow rotation about the z axis. To truly represent all possible straight edged objects we need to allow for rotation about all axes.

B. GRAPHICS

The graphics projection of a model view into 2D coordinates works well. The entire process takes approximately 0.5 seconds of processing time (not including the visibility checking algorithm). As with the model, the graphics projection routines are not generalized to account for camera rotation about all axes. Instead the camera is assumed to only rotate about the z axis. For this reason, we cannot handle cases where the camera is not perpendicular to the z axis (i.e., when Yamabico climbs or descends a ramp).

C. VISIBILITY ALGORITHM

1. Time Comparisons of Different Versions

In order to assess the level of visibility checking which must be applied to a model we have tested three different versions of the sweep algorithm described in Chapter IV. We wish to minimize processing time and receive output which is useful for pattern matching against our camera image. Conclusions are based upon comparisons of processing time[2] and output quality from various configurations within the model of

[2] Processing times are measured by stopwatch in seconds. Times will reflect the total time required to extract lines from the model, conduct visibility checking and project the lines into the final 2d device coordinates.

Spanagel Hall's fifth floor. Figures 6.1a and 6.1b show two of these views as they appear with no visibility checking.

a. Simple 2D Sweep

This algorithm does not take any z information into consideration. All edges of the model are assumed to lay within a single horizontal plane. Figures 6.2a and 6.2b show the sets of visible lines accepted from configurations matching those used in Figures 6.1a and 6.2b respectively.

Total processing time for each view processed under this algorithm averaged 3.0 seconds. As expected, the algorithm is very quick, but it is doubtful that the output is useful for pattern matching. When only two dimensions are considered there can only be one edge visible at any point around the sweep. This obviously hold true for our output. This is a major problem since the closest objects, no matter how small, will occlude all objects which are more distant. Notice that the top of the molding along the left side wall occludes everything behind it (even though the molding is only four inches tall).

Another unexpected problem occurs since lights on the ceiling often fall closest to the camera (in the x-y plane). The camera position in Figure 6.2a had to be adjusted to the left since the position (44,44,44) fell directly underneath a light. At the original position the displayed image is blank, since the edges of the light are accepted by the 2D visibility check but the camera view angle (30 degrees) is not wide enough to allow them to be projected onto the 2D view plane. Even at the adjusted coordinates the entire right side of the display appears blank due to this same light. The same phenomenon is observed in Figure 6.2b.

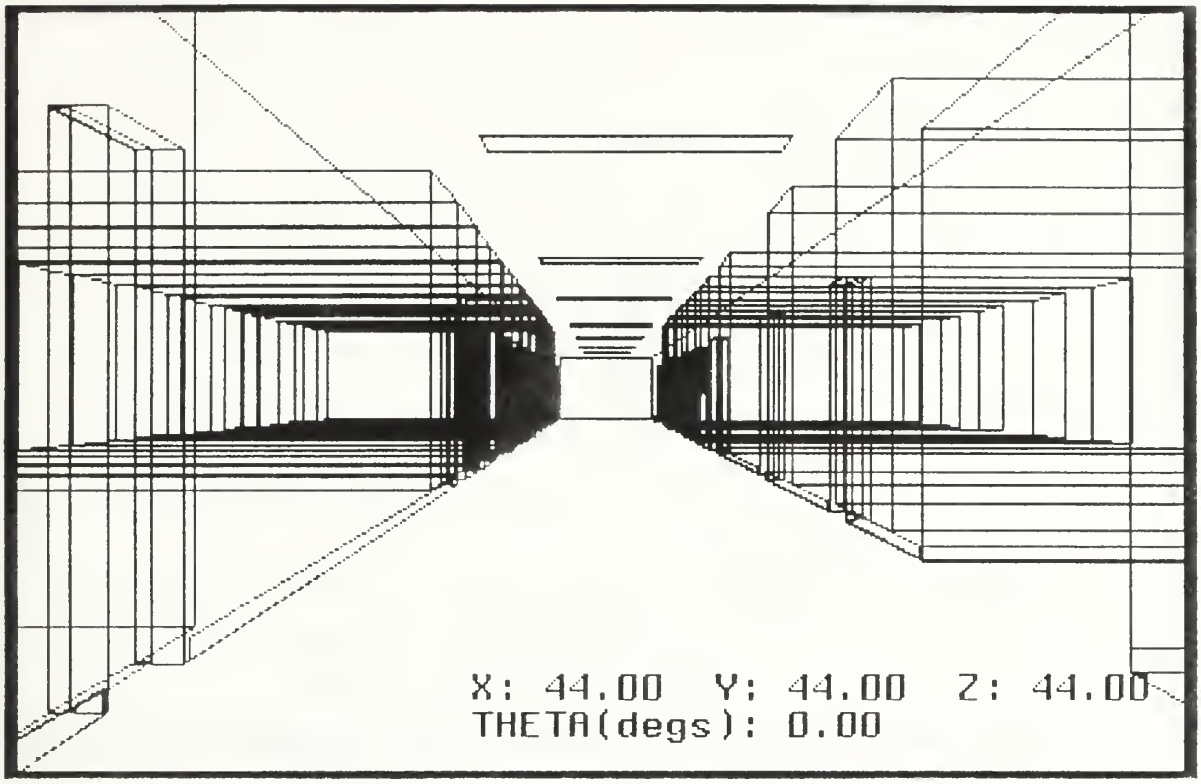


Figure 6.1a

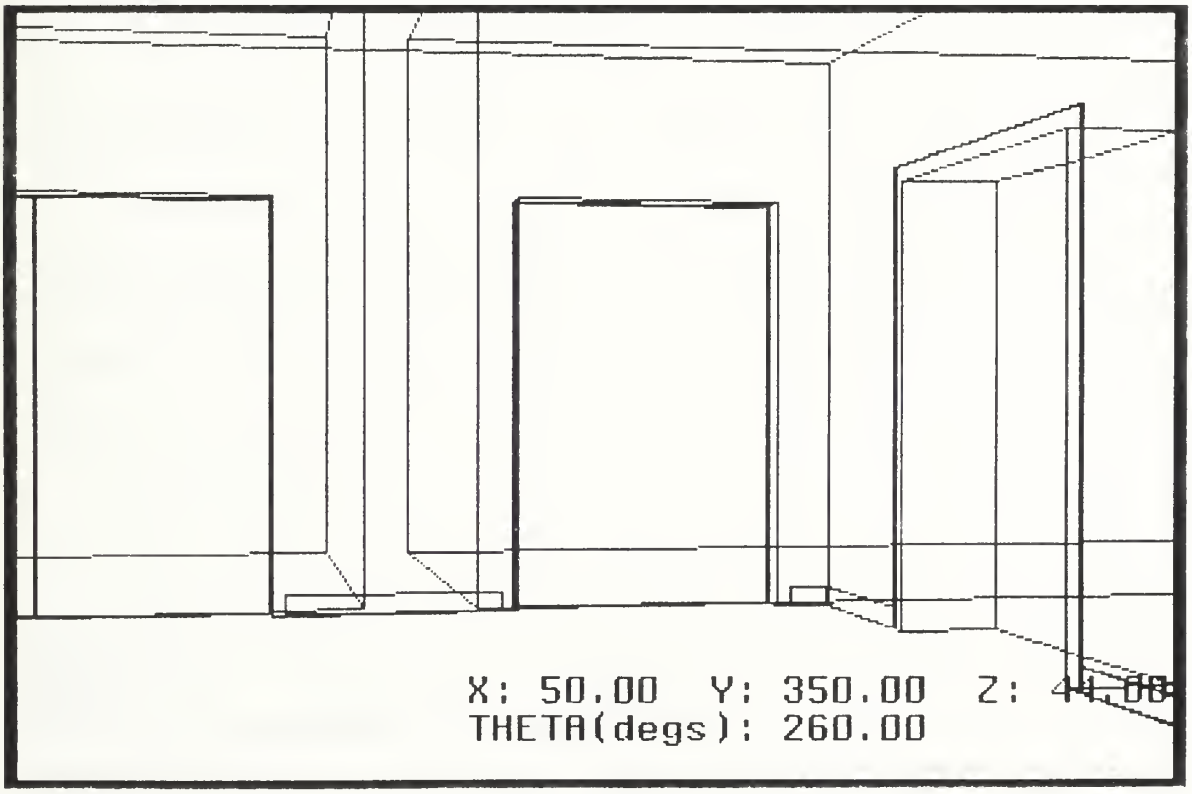


Figure 6.1b

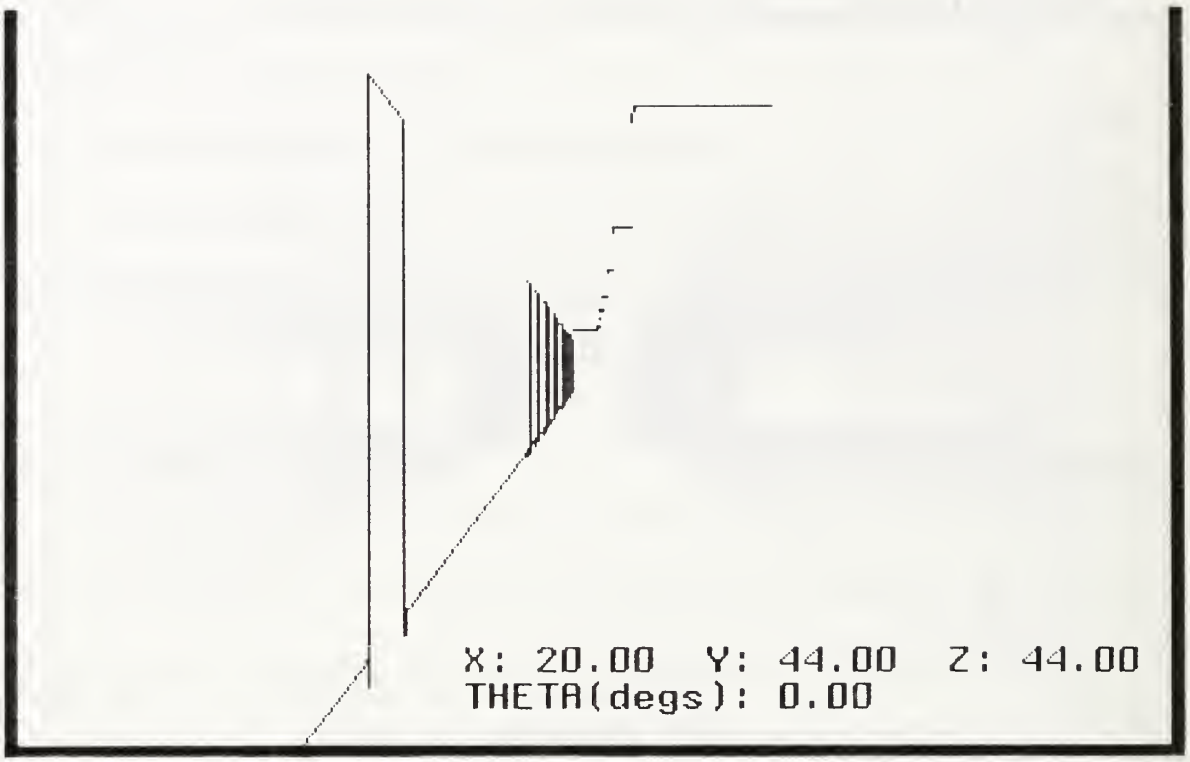


Figure 6.2a

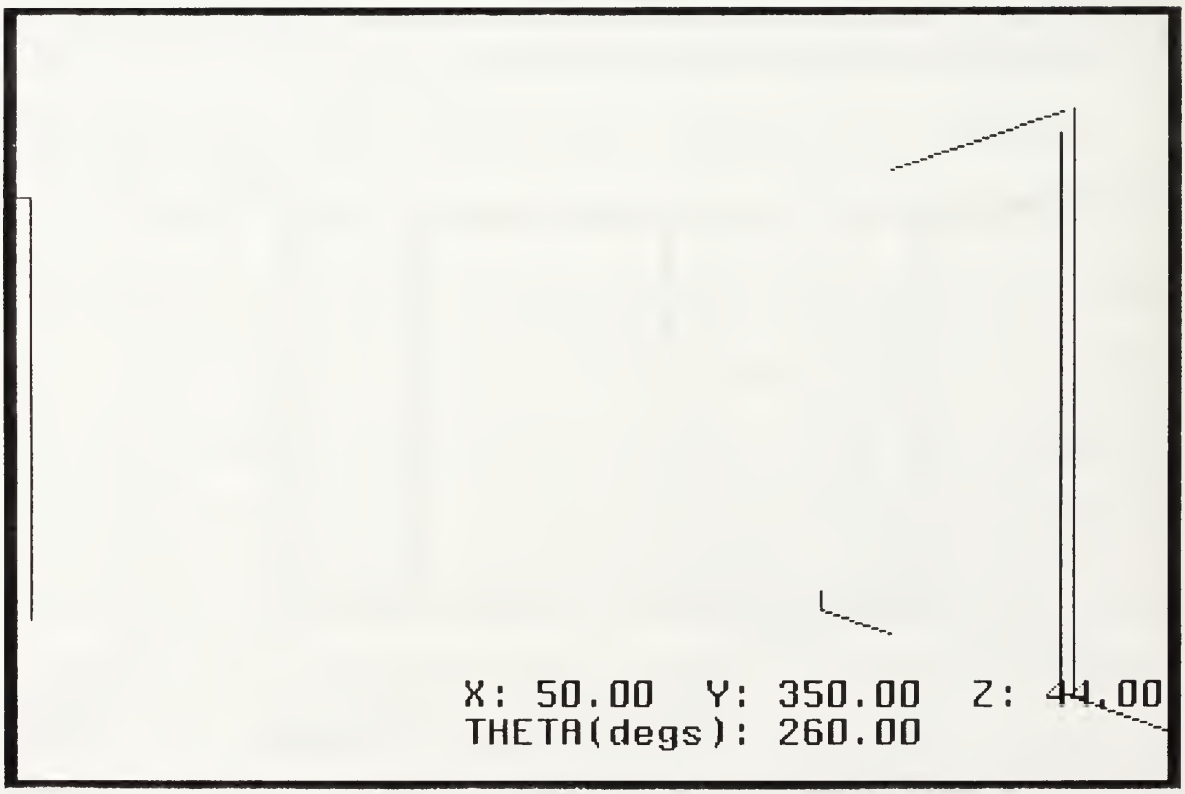


Figure 6.2b

b. Partial 3D Sweep

This algorithm takes z information into consideration but does not account for perspective. The degree of coverage for each edge is calculated when the ccw sweep reaches its first endpoint. To save processing time, this coverage is assumed to remain constant as the sweep progresses along the edge's length. In reality, coverage along the z axis will increase as the distance to the edge decreases (and visa versa). Figures 6.3a and 5.3b show the output from this algorithm.

Total processing time for each view averaged 5.6 seconds. Although somewhat slower than the 2D algorithm, we see a dramatic increase in the output quality. Here the displays are close to what we would expect the camera to see. Unfortunately, numerous short lines are present which should not be and several lines which should be seen are not. Concentrations of these errors increase as we move along the visible edges (away from the first endpoint). This happens because the error between the initial z coverage and actual coverage increases as the sweep progresses along each edge.

The appropriateness of this algorithm depends on how well pattern matching handles extra and missing line segments. If the matching algorithm is capable of disregarding or filtering out lines and allows partial matchings between lines, the partial 3D sweep may be suitable.

c. Full 3D Sweep

This is the complete algorithm described in Chapter IV. Here we recalculate each edge's z coverage and visibility for each angle around the sweep. As mentioned earlier, continuous correction for perspective around the sweep is not feasible. Hopefully,

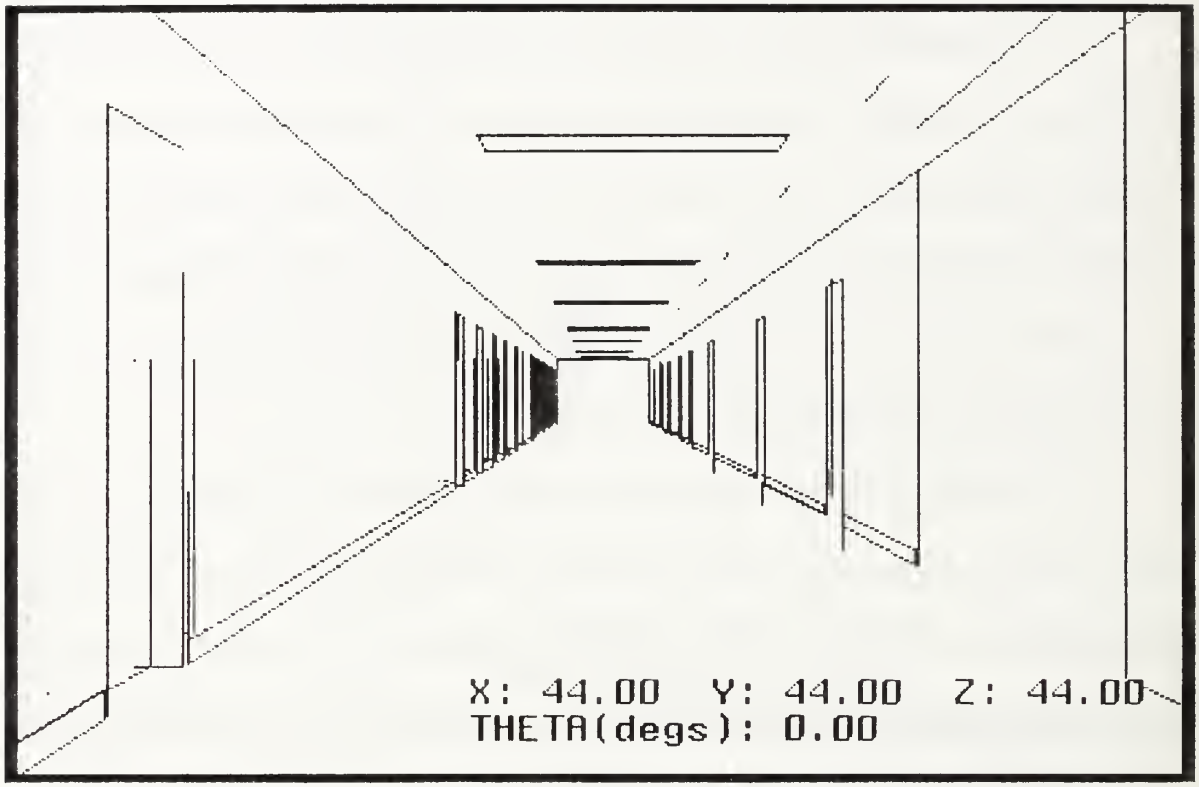


Figure 6.3a

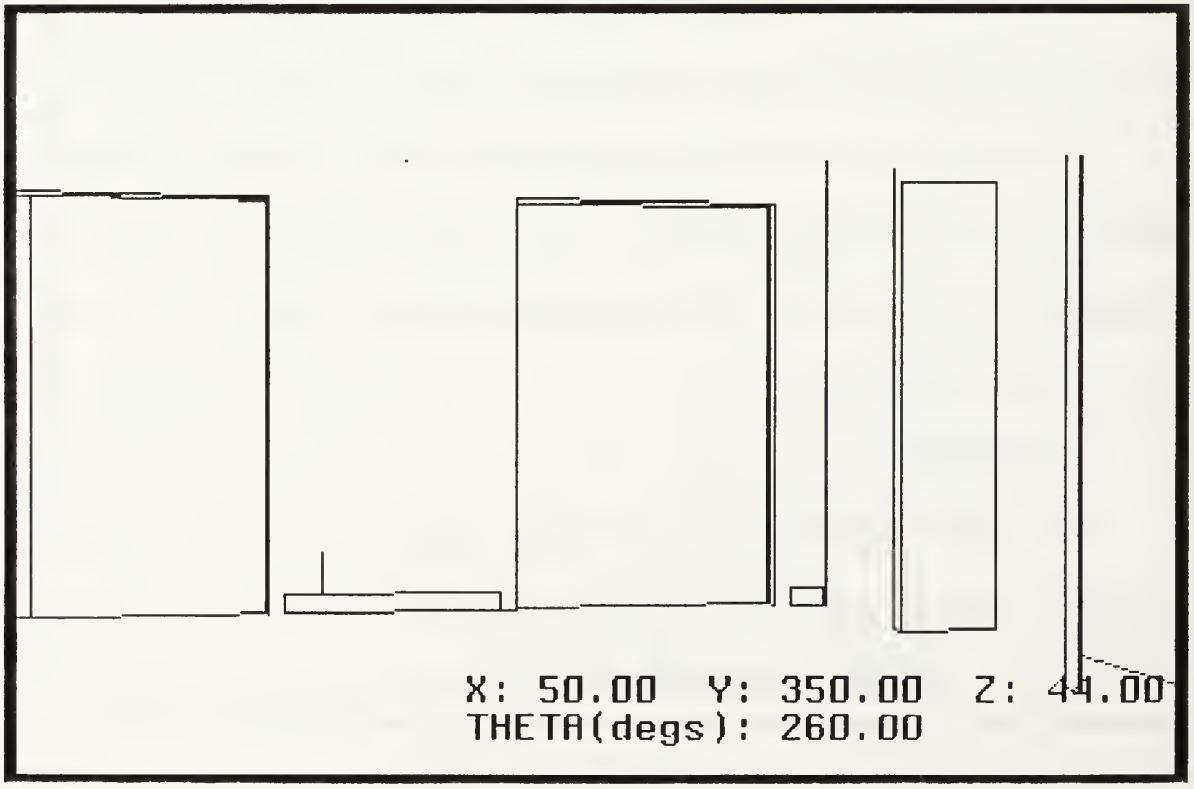


Figure 6.3b

this discrete method of correction provides a reasonable approximation. Figures 6.4a and 6.4b show the output for this algorithm.

Here we find a very disturbing increase in processing time. The average time for the full 3D sweep algorithm is 16.0 seconds. The majority of unwanted lines appearing in Figure 6.3 are no longer present, but several short spikes still occur along some edges. Noticeable 'tails' are present at intersections of leading door edges and door jams. These 'tails' and spikes result from using discreet intervals to re-evaluate z coverage. If a continuous method is developed to account for perspective, these effects should be eliminated.

2. Problems

Our greatest concern with these results lie with the excessive processing time. Sixteen seconds would not seem to readily support real-time processing, but does provide on line support for development of Yamabico's prototype vision system.

The problems due to discreet perspective may interfere with pattern matching. The particular algorithm Kevin Peterson has implemented for our initial position verification system does not seem to be adversely affected.

D. IDEAS FOR FUTURE WORK

1. Data Separation

Building a model through the function calls of a construction file, such as our **5th.h**, is somewhat wasteful. Since the construction file must be compiled and resides in main memory, we gain a large chunk of executable code that is only executed once.

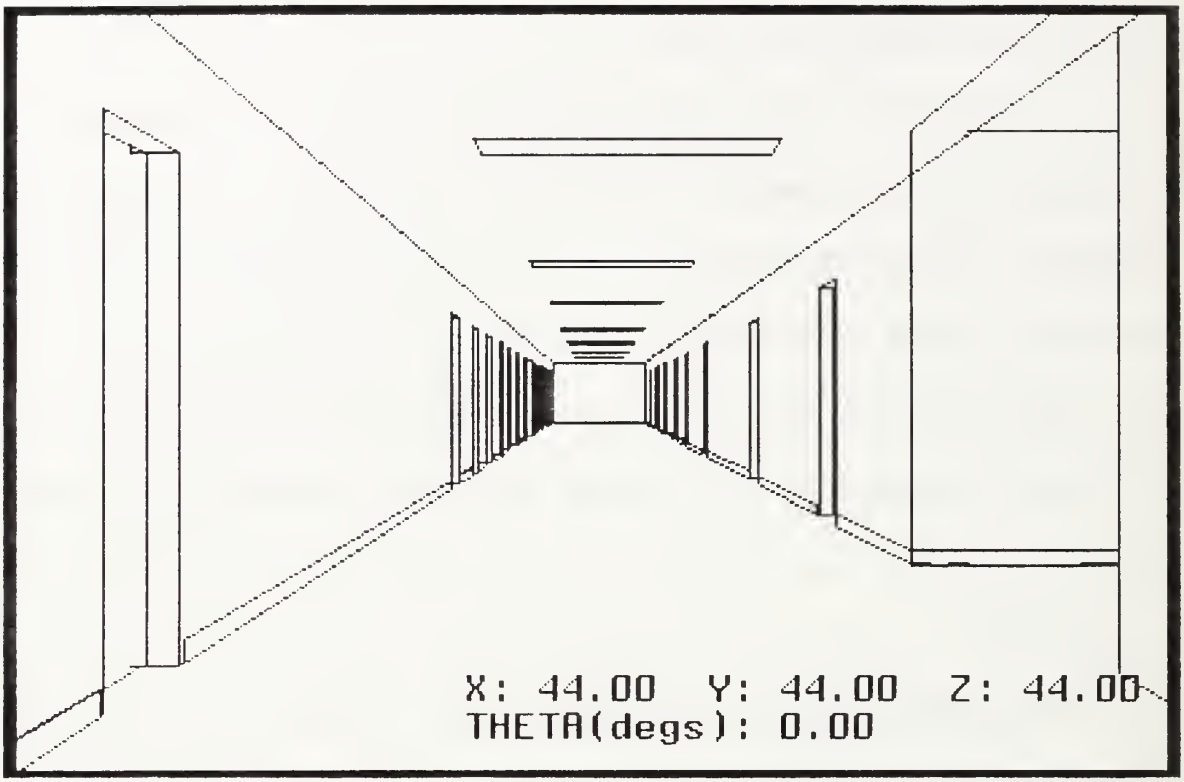


Figure 6.4a

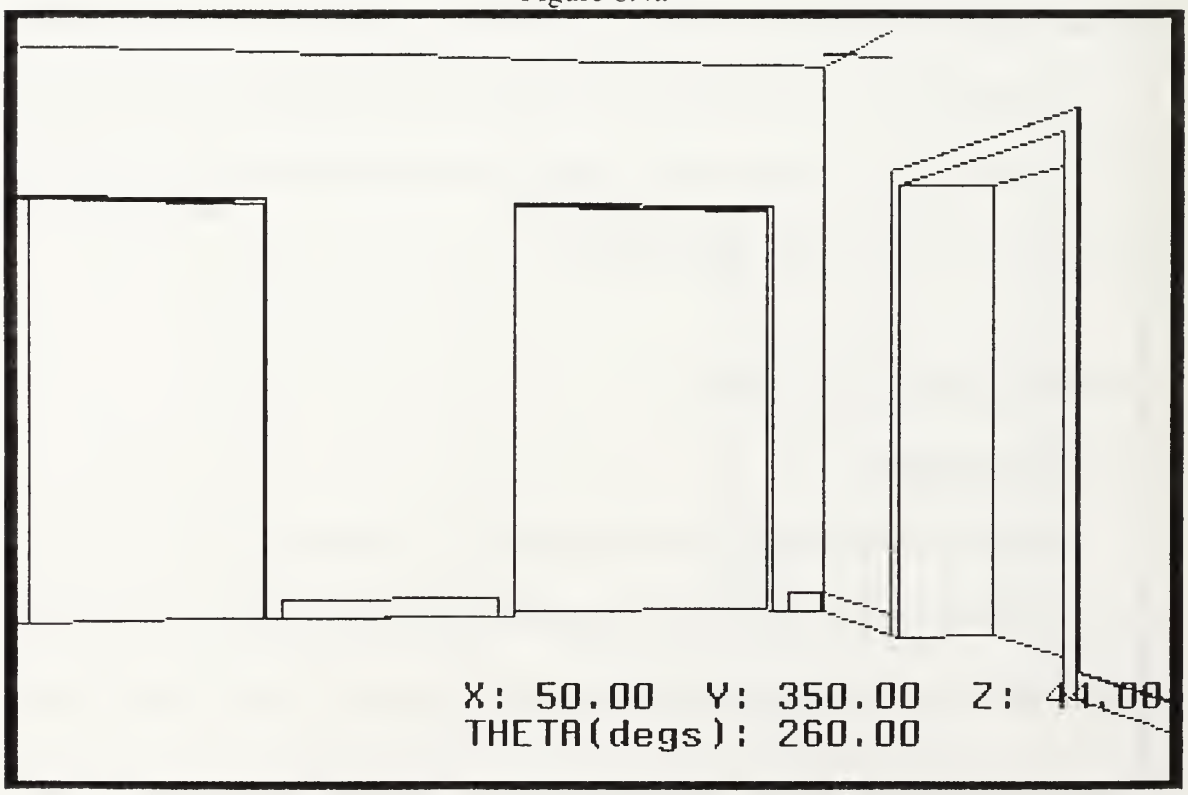


Figure 6.4b

A logical solution to this problem is to separate the model data from the executable file. A text data file could be made to hold the same information that is used in **5th.h**. Unique field separators or a tailored storage hierarchy will need to be employed to mark data. This will allow an iterative function to read the data file and pass the appropriate information to the add functions found in **2d+d.h**.

2. Interactive Interface

If significant modifications and additions to the model are anticipated or if multiple models need to be constructed, development of an interactive interface may be warranted. Such a program could allow entry and modification of model data through a combination of menu driven selections and keyboard entry of coordinates. An interface of this design would also readily support separation of data into a text or binary file.

3. Extend 5th Floor Model

The model constructed by **5th.h** contains most of the major features required for pattern matching, but there are still modifications and additions which need to be made. The most notable modification required is the need to properly model the interior of the office spaces. The current data points within offices are estimates. Due to time constraints, only room 512 is accurately modeled (most other office dimensions are based on this room's measurements). Additions which may prove useful include double doors at both ends of the hall, bulletin boards and chalkboards. The second half of the fifth floor and the offices lying along it also need to be added to our model.

4. Complete Visibility Checking

The previous section identified some problems with our visibility checking algorithm. Investigation into increasing efficiency and correcting for perspective is needed to reduce processing time and increase output accuracy. More accurate output can also be achieved if the method of tracking z coverage for edges is expanded to allow for non-contiguous coverage. This will allow the occlusions of the type shown in Figure 4.7 to be properly represented.

5. Update Graphics Support

Current graphic projection only allows for a camera with three degrees of freedom (mounted perpendicular to the floor and rotating about the z axis). Expanding the functions in **graphics.c** to allow for six degrees of freedom will provide a generalized solution to accommodate rotation of the camera about the x and y axes.

6. Expand Simulator

The simulator can easily be expanded to read configurations from path or mission planning routine output. Restrictions could also be imposed upon speed and turning radius to mirror those of Yamabico. These steps would allow the simulator to be used for mission simulation and analysis, rather than just for inspecting the model.

7. C++ Implementation

The class inheritance, virtual functions and other features available in C++ may prove ideally suited to implementing the 2D+D model structure. Investigation into this possibility is encouraged.

8. Hardware Implementation

All of the work presented in this thesis has been implemented on a personal-iris workstation. The video camera used for collecting data needs to be incorporated into Yamabico's hardware design. It may also be possible to install an additional processor which is dedicated to supporting the vision system. Once hardware is in place, software installation will follow.

VII. USER'S MANUAL

A. INTRODUCTION

This manual discusses the use of the files residing on the ***turing*** personal-iris machine under the /yamabico/model directory. ***Turing*** is located in room 506 of Spanagel Hall at the Naval Postgraduate School, Monterey, California. These files, **2d+d.h, 5th.h, visibility.h, graphics.h, 2d+dsim.h** and **interface.c**, contain functions written in C. The files are part of the vision system being developed for use on-board Yamabico-11 (an autonomous, wheeled robot).

The vision system for Yamabico utilizes a single video camera for input. Information about the robot's operating environment is stored as an asymmetric 2D+D model. The **2d+d.h** file contains the functions, called by **5th.h** for construction and textual display of this model. **Visibility.h** and **graphics.h** provide functions for extracting a view from the model, which may be used in the vision system's pattern matching applications or for graphic display. **2d+dsim.h** contains a basic simulator which allows the user to perform graphic walk-throughs of a modelled world, while the final file, **interface.c**, 'includes' all of the previous files and provides a simple interface through which the user can access and test the various functions.

B. BUILDING A MODEL

1. Construction File

A 2D+D model (Chapter III) is used to represent an orthogonal operating environment. A model is created by successive calls to the various construction functions

in **2d+d.h**. We have created the construction file, **5th.h**, to model the fifth floor of Spanagel Hall. This file consists of one primary construction function, *make_world*, which conducts these calls in the proper order and returns a pointer to the newly built world. All coordinates of our model are entered in inches. The hallway floor is considered to be at a z height of zero inches. The origin of the x-y plane is located at the north east corner of the hall. When looking at the double doors to room 506 this is the lower corner on the right hand side. Orientation with respect to the model is zero (or 360) degrees looking down the x axis (across the hallway) and increases in the ccw direction (Figure 7.1).

Construction files must use the C ***#include 2d+d.h*** command to ensure access to the model support, functions. In turn, the construction file can be *included* to allow its compilation and use by an application program. Each **add** function from **2d+d.h** returns a pointer to the specific structure which is added. The primary construction function takes advantage of this property to keep track of structures within the model and uses the returned pointers as input parameters to future calls. This approach to building a model requires it to be put together in a top down fashion. To accomplish this, a higher level structure such as a polyhedron must be added to the model prior to adding any of its component vertex or instance structures.

2. Declarations

The structure types which are used to build a model are declared at the start of the **2d+d.h** file (A-3). Of these, one world and several polyhedra, polygons and vertices must be declared for use in the construction file. You will notice a significant number

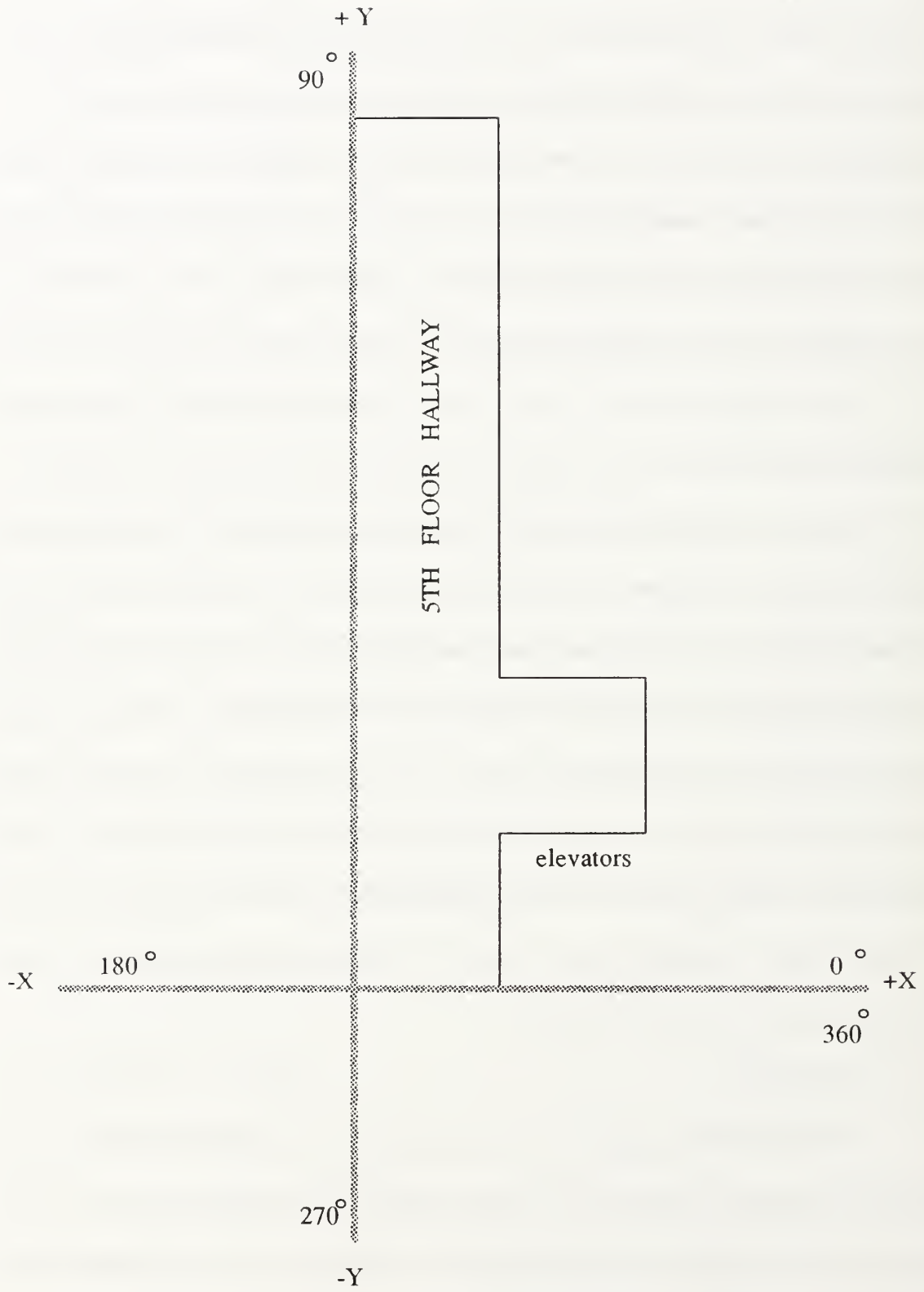


Figure 7.1
Orientation of 5th floor model.

of such declarations in our construction file, **5th.h**. We have adopted a convention in naming pointers to help keep track of relationships between structures. Examples are:

label	representing
H3	Third polyhedron declared and added.
H3P1	First polygon of polyhedron **H3**.
H3P1V1	First vertex of polygon **H3P1**.

3. Building The Model

Once declarations have been made, the first step is to call the *add_world* function in the following format.

```
WORLD *W;
...
W = add_world("name of world",13);
```

PARAMETER	TYPE
1st	character array (30 maximum length)
2nd	integer

The input parameters are simply the label we wish assigned to the world and the number of characters in that label. The function will allocate memory, create an empty world, assign the indicated label to that world and initialize the other fields to indicate the world is currently empty.

Once a world has been created, we can begin to add the various objects which make up the world. Each object is treated as a class of polyhedra. Instances from a class can be instantiated into the model at different positions and with different orientations.

This allows the polyhedron class to be described in a local coordinate system (usually originating at (0,0,0) in the (x,y,z) coordinate system). Recall that a 2D+D model representation contains only horizontal polygons and vertical edges. Groups of these two types will make up each polyhedron. To add an object to the world we use the *add_polyhedron* function.

```
POLYHEDRON *H1;

...

H1 = add_polyhedron("class name",9,W,obstacle,fixed);
```

PARAMETER	TYPE
1st	character array (30 element maximum length)
2nd	integer
3rd	world structure pointer
4th	boolean (0 or 1)
5th	boolean (0 or 1)

Again, the first two parameters represent a class label and its length. The third field must be a pointer to an existing world structure to which this polyhedron should be added. Of the last two boolean parameters, the first indicates if the polyhedron is an obstacle or not. Objects with a 0 in this field are viewed as enclosures and component polygons are expected to have their vertices in cw order. Obstacles will have a 1 here, and component polygons will have ccw lists of vertices. The final boolean tells if the object is fixed in the model or not. Polyhedra like doors should have a 0 in this position to indicate that they can move, while hallways would contain a 1.

When an empty polyhedron has been added to a model, it is important to complete its definition by adding at least one component polygon to it. This is accomplished by a call to *add_polygon*.

```
POLYGON *H1P1;
...
H1P1 = add_polygon(H1,z_value,floor,convex);
```

PARAMETER	TYPE
1st	polyhedron pointer
2nd	float
3rd	boolean (0 or 1)
4th	boolean (0 or 1)

The first parameter must point to a polyhedron which has already been added to the model. The 2nd parameter indicates the height along the z axis at which the vertices of this polygon will be found. **Floor** indicates if the polygon is a floor or ceiling of **H1**. The final field indicates if the polygon is convex or not.

A minimum of three vertices must be added to a polygon to properly define it. Polygons are assumed to be a closed list of vertices so the first vertex should **not** be repeated as the last vertex. It is important to ensure these lists are in cw order for enclosure polyhedra and ccw order for obstacle polyhedra. The *add_vertex* function links vertices into a parent polygon in the order they are added.

VERTEX *H1P1V1;

...

H1P1V1 = add_vertex(H1P1,x,y);

PARAMETER	TYPE
1st	polygon structure pointer
2nd	float
3rd	float

Each vertex must be added to an existing polygon pointer. The last two parameters denote the x and y coordinates at which the vertex is located.

Once all of the polygons making up a polyhedron have been added, we need to indicate the location of vertical edges. The *add_edge* function accepts two vertices as input and creates a pointer from the first to the second which represents a vertical edge.

add_edge(P1V1,P2V2);

PARAMETER	TYPE
1st	vertex structure pointer
2nd	vertex structure pointer

Vertical edges should only be placed once between each pair of vertices. Each vertex must be from different polygons residing in the same polyhedron structure. Notice that there is no value returned.

To properly identify which ceilings enclose a floor we must use the *add_ceiling* function. Here the two parameters are polygons. The first is a floor and the second a ceiling. Ceiling associations are maintained as a list which allows more than one ceiling

to cover a floor. Although it is not required, vertical edges will usually be present between floors and their ceilings.

add_ceiling(H1P1,H1P2);

PARAMETERS	TYPE
1st	polygon structure pointer
2nd	polygon structure pointer

After all the polygons, vertices, vertical edges and ceiling associations have been added to a polyhedron class, we can instantiate the object into our world model using *add_instance*. This conserves storage space within the model and allows for easy addition of the identical objects often found indoors (i.e., doors and lights).

add_instance("label",5,H1,X,Y,Z,PIVOT_X,PIVOT_Y,ROT);

PARAMETER	TYPE
1st	character array (30 maximum)
2nd	integer length of label array
3rd	polyhedron structure pointer
4th-6th	float, (x,y,z) position in model coordinates
7th-8th	float, (x,y) position of polyhedron's pivot point in local coordinate system
9th	float, number of degrees to rotate object about pivot point

When an instance needs to be used, all vertices are translated so the pivot point becomes the local origin. Then the object is rotated about the z axis ROT degrees and placed at the (X,Y,Z) coordinates of the world.

It is not mandatory for instances of a polyhedron class to be added to the model. When no instances are added, views from the model will not include the polyhedron.

The construction file, **5th.h**, is solely responsible for building our world model. To modify the model, we can simply edit the function calls made in this file. The use of instances makes adding new objects very straight forward and allows the orientation of existing, movable objects (i.e., doors) to be altered.

C. CHECKING VISIBILITY

The file **visibility.h (A-14)** contains the functions which conduct visibility checking on a 2D+D model. When a view from the model is being extracted for use in pattern matching or graphic display, lines that are not visible should be filtered out. The *conduct_visibility_sweep* function is called to provide the list of lines which can be seen from a specific point in the model[3]. These lines do not take the type of camera or its orientation into consideration. This means that even lines behind the camera will be returned as visible. The next section tells how to get rid of these unwanted lines. Format of function call:

---

[3] The *get_view* function from the **graphics.h** file calls this function automatically. If you are retrieving a view from the model through this function a separate call to *conduct_visibility_sweep* need not be made.

```
LINE_HEAD *LH;

...

LH = conduct_visibility_sweep(W,X,Y,Z);
```

The LINE_HEAD type allows output from the visibility sweep to be separated into two lists. One list is of vertical lines and the other contains non-vertical lines. Input consists of a pointer to the world being checked and the camera position in (x,y,z) coordinates.

D. GRAPHIC PROJECTION FROM MODEL

The **graphics.h** file contains the function *get_view*. This function calls the *conduct_visibility_sweep* function, and removes those lines (and partial lines) which cannot be seen by the camera. The resulting set of visible 3D lines is then projected onto a 2D window and mapped to a set of final device coordinates. Output is a **LINE_HEAD** pointer which consist of two lists: a list of 2D vertical lines and a list of 2D non-vertical lines. Vertical lines are sperate to help in the pattern matching process. The format for calling *get_view* is:

```
LINE_HEAD *LH;

...

LH = get_view(W,X,Y,Z,ORIENTATION,FOCAL_LENGTH);
```

PARAMETER	TYPE
1st	world structure pointer
2nd-4th	floating point (x,y,z) coordinates
5th	float, camera orientation in degrees

6th float, focal length of camera

A group of definitions can be found at the top of **graphics.h** (A-37). These values are integral to proper projection. The **CCD** is set to two thirds of an inch, which is the physical dimension of our camera's sensing element. This value is used as the 2D projection window dimension (CCD by CCD square), and when used with the input FOCAL_LENGTH determines the viewing angle of the camera. The variables MAX_X and MAX_Y denote the limits of the desired output coordinates to which the 2D lines should be mapped. If the camera, camera zoom setting or output device is changed the user should inspect these variables to determine if their definitions are in need of revision.

E. SIMULATOR

A basic simulator resides in the file **2d+dsim.h**. The primary function, *simulator*, is passed a world structure pointer. The user is queried for the starting configuration within this world ((x,y,z) coordinates and orientation). This simulator is mouse driven. The left and right buttons are used to turn to the left and right respectively. The center button brings up a menu of options (Figure 7.2). To move past the initial view the 'start/restart' option must be selected. The speed of the walk-through starts at zero and can be increased or decreased by choosing the appropriate menu option. Speed can be reset to zero through the 'stop' option, while 'pause' suspends execution until 'start/restart' is selected.

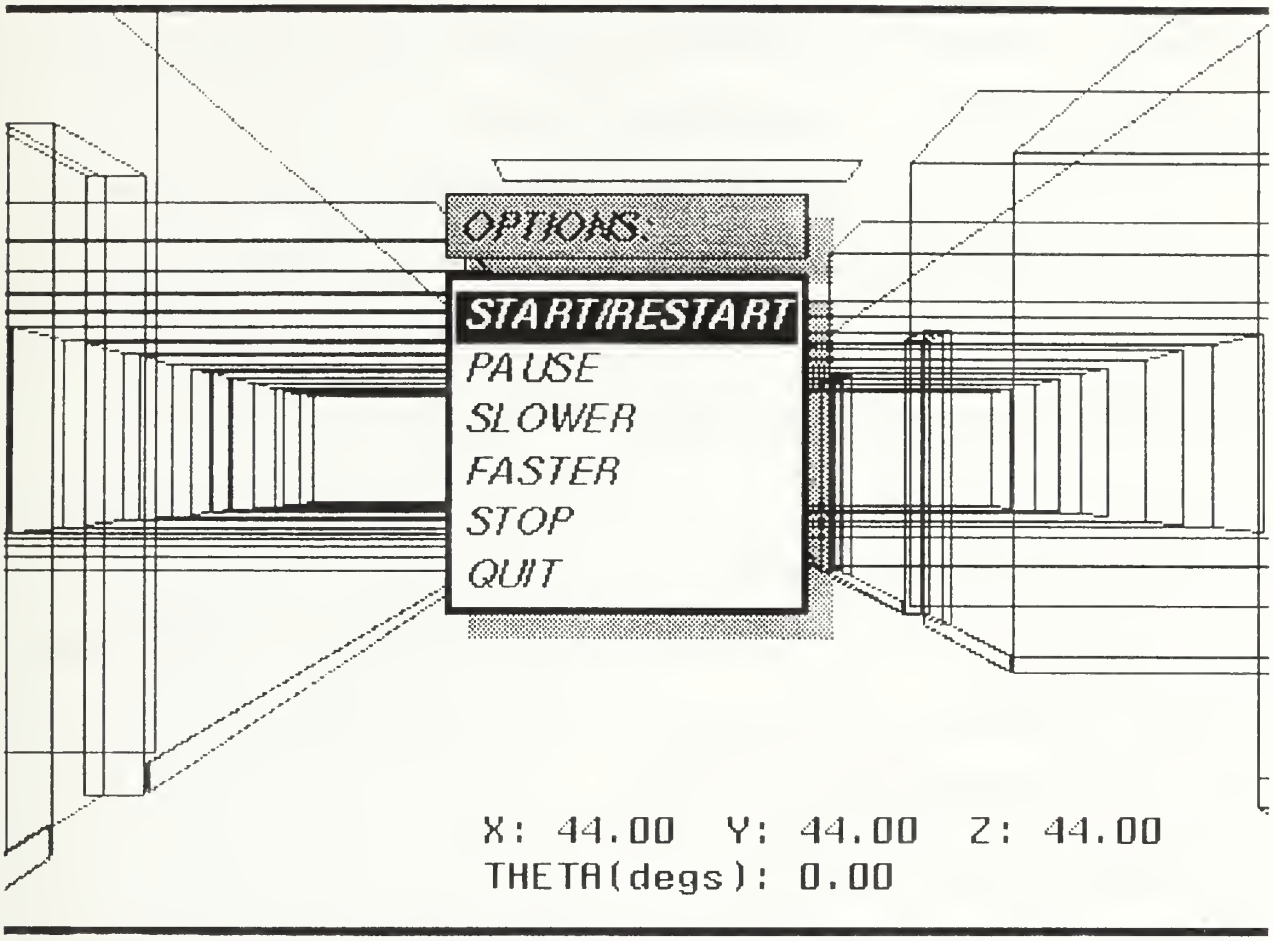

OPTIONS:
START/RESTART
PAUSE
SLOWER
FASTER
STOP
QUIT
X: 44.00 Y: 44.00 Z: 44.00
THETA(degs): 0.00

Figure 7.2

The simulator currently uses the *get_view* function to retrieve the set of lines seen from each configuration as we move through the model. If the simulation speed is too slow the *get_full_view* function can be substituted for both the calls (A-51). This will draw all lines from the model to the screen. It may be useful to make this change when inspecting entry of new objects to the model. Configurations are displayed across the bottom of the display window. Coordinates are shown in inches while the orientation is in degrees.

F. FINDING A POLYHEDRON

The function *find_ph* (A-13) is provided to display a text listing of a polyhedron and all of its instances.

find_ph(LABEL,W);

PARAMETER	TYPE
1st	array of characters (maximum length of 30)
2nd	world structure pointer

Polyhedra structures in the world, W, are searched to find the class named LABEL. If the need arises, the user can easily modify this function to return a pointer to the polyhedron that is found.

G. DEALLOCATING MEMORY

It is important to release memory which is no longer needed back to the operating system. We provide two functions, *free_world* and *free_lines* to release memory held in a world and list of lines respectively.

free_world(W);

free_lines(LH);

Both functions use the C *free* function to release each substructure held in all lists that make up the world or line list.

H. TROUBLESHOOTING

The following suggestions are made to help track down the cause of problems which may arise when using this software. If the user wishes to gain a more thorough understanding of why the 2D+D model was selected or how the underlying structures interrelate, they are referred to the first five chapters of this thesis and to the code and documentation found in Appendix A.

PROBLEM	POSSIBLE CAUSE(S)
1. missing vertical lines	-*add_edge* calls not made properly in construction file
2. objects missing	-*add_instance* not called in construction file
3. objects facing wrong direction	-rotation angle in *add_instance* incorrect
4. view stretched or shrunk	-camera or camera zoom setting changed causing incorrect definition of CCD or focal length

5. views do not match actual	-incorrect CCD or focal length camera image -device coordinates (MAX_X,MAX_Y) incorrect
6. **all** model lines seen	-call to *get_full_view* vice *get_view* being made from simulator
7. cannot turn simulator	-**start** menu option not yet selected
8. turns too much/little	-adjustment needed on increment of left and right cases found in *simulate* function
9. for each mouse click	-adjustment needed on increment of speed changes too slow/fast speed options (3&4) in *processmenuhit* function. (File **2d+dsim.h**)
10. labels for model structures	-global variable MAX_LEN in **2d+d.h** truncated too short needs to be increased
11. core dumps during execution	-a polygon used in the model has no vertices added to it -a structure in construction file is used prior to being added to the model

VIII. REFERENCES AND BIBLIOGRAPHY

A. REFERENCES

1. Galbiati, Louis J., *Machine Vision and Digital Image Processing Fundamentals*, Prentice Hall, 1990.

2. Shirai, Yushiaki, *Three-Dimensional Computer Vision* Springer-Verlag 1987.

3. Fairhurst, Michael C., *Computer Vision for Robotic Systems: an Introduction*, Prentice Hall, 1988.

4. Therrien, Charles W., *Decision, Estimation, and Classification: an Introduction to Pattern Recognition and Related Topics*, Wiley, 1989.

5. Thorpe, Charles W. (ed.), *Vision and Navigation: The Carnegie Mellon Navlab*, Kluwer Academic Publisher, 1990.

6. Robert, L., Vaillant, and Schmitt, "3-D-Vision-Based Robot Navigation: First Steps," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

7. Deriche, R., and Faugeras, O., "Tracking Line Segments," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

8. Crowley, J. L., and Stelmaszyk, P., "Measurement and Integration of 3-D Structures by Tracking Edge Lines," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

9. Holder, D., and Buxton, H., "SIMD Geometric Hashing," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

11. Andersen, J. D., "Combinatorial Characterization of Perspective Projections From Polyhedral Object Scenes," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

12. Leavers, V. F., "The Dynamic Generalized Hough Transform," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

13. Foley, James D., and others, *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company, 1990.

B. BIBLIOGRAPHY

1. Barr, A., Cohen, P. R., and Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, vol. IV, Addison-Wesley Publishing Company, Inc., 1989.

2. Gardner, J., *From C to C: An Introduction to ANSI Standard C*, Harcourt Brace Jovanovich, Publishers and its Subsidiary, Academic Press, 1989.

3. Kelley, A., and Pohl, I., *A Book on C: Programming in C*, 2nd ed., The Benjamin/Cummings Publishing Company, Inc., 1990.

4. Lowe, D. G., "Stabilized Solution for 3-D Model Parameters," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

5. Provan, G. M., "An Analysis of Knowledge Representation Schemes for Higher Level Vision," *European Conference on Computer Vision Proceedings*, 23-27 April 1990.

6. Vieville, T., "Estimation of 3-D Motion and Structure From Tracking 2-D Lines in a Sequence of Images," *European Conference on Computer Vision Proceedings*, 23-27 April 19.

APPENDIX A

```
#include <math.h>
#include <stdio.h>
#include <device.h>
#include <gl.h>
#include "2d+.c"
#include "5th.c"
#include "my_graphics.c"
#include "visibility.c"
/*#include "vis2d.c"*/                    /*2d and partial 3d visibility checks*/
/*#include "vis3d.c"*/
#include "2d+sim.c"

/*****************************************************************
  FILENAME: interface.c
  AUTHOR:   LT James Stein
  DATE:     Mar 1992
  Project:  Thesis, supporting Yamabico's vision system
  COMMENTS: This file has been written as an interface to help test the various functions needed to support the 2d+ model. The construction
file 5th.h is used to build the 2d+ model of Spanagel Hall's 5th floor into memory. Once this is done the user can choose to dump a test listing
of the model to the screen, search for a polyhedron class by name, get a view from the model, or conduct a graphics walkthru of the model.
*****************************************************************\


/*  constants */
#define Pl                      3.141592653589793
#define MAX_LEN         30


void main()
{
  WORLD *W=NULL;
  char PH_LABEL[MAX_LEN], c;
  int OPTION =1;
  int i, PH_NUM;
  float X,Y,Z,ORIENT,FOCAL_LENGTH=1.24; /*FOCAL_LENGTH must be adjusted for*/
  LINE_HEAD *LIST=NULL:                                 /*different cameras or zoom settings*/

  W = make_world();    /*from file 5th.c*/
  while (OPTION>0) {
    printf("\n\n\n1.Find a polyhedral ");
    printf("\n2.Display world (text listing)");
    printf("\n3.Conduct graphics walkthru (iris terminal required)");
    printf("\n4.Get view for pattern matching");
    printf("\n\nChoose one (0 to quit): ");
    scanf("\n%d",&OPTION);
    switch (OPTION) {
    case 1:         /* find a ph */
      for (i=0;i<MAX_LEN;++i) {  /*clear old label*/
            PH_LABEL[i]=' ';
      }
      printf("\nPlease enter the label of the polyhedron you wish to see");
      printf("\n(20 char max): ");
      scanf("\n%s",PH_LABEL);
      printf("\nIndexing on label: (%s)\n",PH_LABEL);
      find_ph(PH_LABEL,W);  /*file 2d+.h*/
      break;
    case 2:   /*dump text listing of world to screen*/
      display_world(W);   /*file 2d+.h*/
      break;
```

```
    case 3:   /*conduct graphics simulation of walkthru*/
      simulate(W);          /*file 2d+sim.h*/
      break;
    case 4:   /*get a set of lines making up a view in the model*/
      printf("\n\nEnter coordinates for position of camera:\nX: ");
      scanf("\n%f",&X);
      printf("\nY: ");
      scanf("\n%f",&Y);
      printf("\nZ: ");
      scanf("\n%f",&Z);
      printf("\nOrientation (0.0 is down y-axis): ");
      scanf("\n%f",&ORIENT);
      LIST=get_full_view(X,Y,Z,ORIENT,W,FOCAL_LENGTH);/*file graphics.h*/
      if (LIST)
            free_lines(LIST);   /*deallocate the memory used*/
      break;
    case 0:
      printf("Exiting program\n\n");
      break;
    default:
      printf("Invalid choice!!!");
    }  /* end case statement */
  }  /* end while loop */
  free_world(W);   /* deallocate world memory */
} /* end main procedure: make_world */
```

```
/*****************************************************************************

  FILENAME: 2d+d.h
  AUTHOR:   LT James Stein
  CONTENTS: 2d+ model support tools (for building, displaying, searching,
          and deallocating a model)
  DATE:     Mar 1992
  COMMENTS: A 'world' consists of a list of polyhedrons (PH) Each PH is in
turn a list of polygons (PG). Each PG is a list of VERTICIES which contain
the X,Y, and Z coordinates of that point in the world.
  File 5th.h is an example construction file which uses these functions to
build a model of the 1st half of Spanagel Hall's 5th floor.

*****************************************************************************/


/*  constants */
#define PI                      3.141592653589793
#define MAX_LEN         30

/* typedefs: Define structures to be used for representing a 3-d world */

typedef struct vertex {
  float X,Y;
  struct vertex
    *NEXT, *PREV,
    *VERT_EDGE;
} VERTEX;

/* WHERE:  VERT_EDGE = pointer to upper vertex of vertical edge

---------------------------------------*/

typedef struct poly_link {
  struct polygon *REF_POLY;
  struct poly_link *NEXT, *PREV;
} POLY_LINK;

/*---------------------------------------*/
typedef struct polygon {
  int DEGREE, C_DEGREE, FLOOR, CONVEX;
  float Z_VALUE;
  VERTEX *VERTEX_LIST;
  POLY_LINK *CEILING_LIST;
  struct polygon
    *NEXT, *PREV;
} POLYGON;

/* WHERE:          DEGREE = # of vertices
                   FLOOR, CONVEX = booleans
                   Z_VALUE = local Z position poly located at
                   CEILING_LIST, FLOOR_LIST = list of associated poly's
---------------------------------------*/

typedef struct instance {
  char NAME[MAX_LEN];
  float X, Y, Z, ROTATION,
       PIVOT_X, PIVOT_Y;
  struct instance *NEXT, *PREV;
} INSTANCE;
```

```
/* WHERE:        NAME = something like "rm501"
                 X, Y, Z = position to instantiate PH into world
                 ROTATION = degrees to rot about Z axis

----------------------------------------*/

typedef struct polyhedron {
  char CLASS[MAX_LEN];
  int DEGREE, I_DEGREE, OBSTACLE, FIXED;
  POLYGON *POLYGON_LIST;   /*ordered by Z value*/
  INSTANCE *INSTANCE_LIST; /*ordered by Z value*/
  struct polyhedron *NEXT, *PREV;
} POLYHEDRON;

/* WHERE:        CLASS = general name like 'door'
                 DEGREE = # of polygons
                 OBSTACLE and FIXED = booleans
                 CEILING_LIST, FLOOR_LIST = list comprise all polygons
                 INSTANCE_LIST = all tranformations of object into world

----------------------------------------*/

typedef struct world {
  char NAME[MAX_LEN];
  int DEGREE;
  POLYHEDRON *POLYHEDRON_LIST;
} WORLD;

/* WHERE:        NAME = label for world
                 DEGREE = number of object representations
                 POLYHEDRON_LIST points to them
*/

/*************************************************************************

 The following routines are called to allocate memory for a structure
 (WORLD, POLYHEDRON, POLYGON, or VERTEX). Pointers are initialized to NULL
 and the DEGREE field is set to 0;

*************************************************************************/

WORLD *create_world()
{
  WORLD *W;
  int i;

  /* allocate memory for a world */
  if((W = (WORLD *)malloc(sizeof(WORLD))) == NULL) {

    printf("\ncannot create a world\n");
  }
  /* initialize fields */
  W->DEGREE = 0;
  W->POLYHEDRON_LIST = NULL;
  for (i=0; i<MAX_LEN; ++i) {
    W->NAME[i]=' ';
  }
  return(W);
}


/*----------------------------------------------------------------------*/
```

```
POLYGON *create_polygon()
{
  POLYGON *P;

  /* allocate memory for a polygon */
  if((P = (POLYGON *)malloc(sizeof(POLYGON))) == NULL) {
    printf("\cannot create a polygon");
  }

  /* initialize fields */
  P->DEGREE = 0;
  P->Z_VALUE= 0.0;
  P->VERTEX_LIST = NULL;
  P->CEILING_LIST= NULL;
  P->NEXT = NULL;
  P->PREV = NULL;

  return(P);
}


INSTANCE *create_instance()
{
  INSTANCE *I;
  int i;

  I= (INSTANCE *)malloc(sizeof(INSTANCE));
  for (i=0; i<MAX_LEN; ++i) {
    I->NAME[i]=' ';
  }
  I->NEXT = NULL;
  I->PREV = NULL;
  return I;
}


/*-----------------------------------------------------------------*/


POLY_LINK *create_poly_link() {

  POLY_LINK *P;
  P=(POLY_LINK *)malloc(sizeof(POLY_LINK));
  P->REF_POLY = NULL;
  P->NEXT = NULL;
  P->PREV = NULL;
  return P;
}


/*-----------------------------------------------------------------*/

POLYHEDRON *create_polyhedron()
{
  POLYHEDRON *P;
  int i;

  P=(POLYHEDRON *)malloc(sizeof(POLYHEDRON));
  for (i=0; i<MAX_LEN; ++i) {
    P->CLASS[i]=' ';
  }
```

```
  P->DEGREE=0;
  P->POLYGON_LIST=NULL;
  P->NEXT=NULL;
  P->PREV=NULL;
  P->INSTANCE_LIST=NULL;
  return P;
}  /* end create_polyhedron */


/*---------------------------------------------------------------*/


VERTEX *create_vertex()
{
  VERTEX *V;

  V=(VERTEX *)malloc(sizeof(VERTEX));
  V->NEXT=NULL;
  V->PREV = NULL;
  V-> VERT_EDGE =NULL;
  return V;
}


/*************************************************************************

  The following routines are used for memory deallocation. Each type of
list is stepped through to free it's component structures. Higher level
structures call the free routine for the next lower level to deallocate
side lists (i.e. free_world calls free_polyhedron).

*************************************************************************/

void free_pg(PG)
     POLYGON *PG;
{
  VERTEX *NEXT_V, *TRASH;
  POLY_LINK *NEXT_LINK, *TRASH2;

  NEXT_V=PG->VERTEX_LIST;   /*free vertex list*/
  while (NEXT_V) {
    TRASH=NEXT_V;
    NEXT_V=NEXT_V->NEXT;
    free(TRASH);
  }
  NEXT_LINK=PG->CEILING_LIST;
  while (NEXT_LINK) {          /*free links used to reference ceilings*/
    TRASH2=NEXT_LINK;
    NEXT_LINK=NEXT_LINK->NEXT;
    free(TRASH2);
  }
  free(PG);  /*free parent polygon structure */
}   /* end free_pg */

/*---------------------------------------------------------------*/

void free_ph(PH)
     POLYHEDRON *PH;
{
  POLYGON *NEXT_PG, *TRASH;
  INSTANCE *NEXT_I, *TRASH2;
```

```
  NEXT_PG=PH->POLYGON_LIST;
  while (NEXT_PG) {          /*free the list of polygons*/
    TRASH=NEXT_PG;
    NEXT_PG=NEXT_PG->NEXT;
    free_pg(TRASH);
  }
  NEXT_I = PH->INSTANCE_LIST;
  while(NEXT_I) {            /*free the list of instances*/
    TRASH2= NEXT_I;
    NEXT_I= NEXT_I->NEXT:
    free(TRASH2);
  }
  free(PH);   /* release parent structure */
}   /* end free_ph */

void free_world(W)
     WORLD *W;
{
  POLYHEDRON *NEXT_PH, *TRASH:

  if (W) {
    NEXT_PH=W->POLYHEDRON_LIST;
    while (NEXT_PH) {            /*free the list of polyhedra*/
      TRASH=NEXT_PH;
      NEXT_PH=NEXT_PH->NEXT;
      free_ph(TRASH);
    }
  }
  free(W);
}   /* end free_world */



/*********************************************************************

  The next group of functions is used to display the world. A single
polygon, a single polyhedron, or the entire world can be displayed.
Display is in text format to the standard output device.

*********************************************************************/



void display_pg(PG)
     POLYGON *PG;
{
  POLYGON *NEXT_PG;
  POLY_LINK *NEXT_C;
  VERTEX *NEXT_V;
  int V_NUM=1, PRINTED=0;

  printf("\nDEGREE: %d FLOOR: %d  Convex: %d ",PG->DEGREE,PG->FLOOR,
            PG->CONVEX);
  printf("\nZ = %.2f:\n",PG->Z_VALUE);
  NEXT_V=PG->VERTEX_LIST;
  while (NEXT_V) {
    if (PRINTED>3) {     /* three vertices per line*/
      printf("\nV#%d(%.2f,%.2f) ",V_NUM,NEXT_V->X,NEXT_V->Y);
      PRINTED=1;
    }
    else {
      printf("V#%d(%.2f,%.2f) ",V_NUM,NEXT_V->X,NEXT_V->Y):
```

```
      PRINTED++;
    }
    NEXT_V=NEXT_V->NEXT;
    V_NUM++;
  } /*end while */
  if (PG->FLOOR==1)
    printf("\nAssociated ceilings (%d): ",PG->C_DEGREE);
  NEXT_C= PG->CEILING_LIST;
}   /* end display_pg */

/*-----------------------------------------------------------------*/

void display_ph(PH)
    POLYHEDRON *PH;
{
  POLYGON *NEXT_PG;
  int PG_NUM,F_CNT=1,C_CNT=1,I_CNT=1;
  char dummy;
  INSTANCE *NEXT_I;

  printf("\nPOLYHEDRON (%s):\n Obstacle: %d  Fixed: %d \n",
            PH->CLASS,PH->OBSTACLE,PH->FIXED);
  printf("\nComponent polygons (%d):\n ",PH->DEGREE);
  NEXT_PG=PH->POLYGON_LIST;
  printf("\n\nList of floors:");
  while (NEXT_PG) {
    if (NEXT_PG->FLOOR==1) {
      printf("\n\nFLOOR# %d ",F_CNT);
      display_pg(NEXT_PG);          /*display floor polygons*/
      F_CNT++;
    } /* end if */
    NEXT_PG=NEXT_PG->NEXT:
  } /* end while */
  NEXT_PG=PH->POLYGON_LIST;
  printf("\n\nList of ceilings:");
  while (NEXT_PG) {
    if (NEXT_PG->FLOOR==0) {
      printf("\n\nCEILING # %d ",C_CNT);
      display_pg(NEXT_PG);              /*display ceilings*/
      C_CNT++;
    } /* end if */
    NEXT_PG=NEXT_PG->NEXT;
  } /* end while */
  printf("\n\nThe following instantiations of this polyhedron exist:");
  fflush(stdout);
  if (PH==NULL) {
          printf("\n\ndereferencing null pointer in display_ph\n\n");
          fflush(stdout);
  }
  NEXT_I=PH->INSTANCE_LIST;
  while(NEXT_I) {
    printf("\n\nInstance #%d (%s): ",I_CNT,NEXT_I->NAME);
    fflush(stdout);
    printf("\nLocation: (%.2f,%.2f,%.2f)",NEXT_I->X,NEXT_I->Y,NEXT_I->Z);
    fflush(stdout);
    printf("Rotated: %.2f degrees about point: (%.2f,%.2f)\n",
            NEXT_I->ROTATION,NEXT_I->PIVOT_X,NEXT_I->PIVOT_Y);
    fflush(stdout);
    I_CNT++;
    NEXT_I=NEXT_I->NEXT;
  } /* end while */
}   /* end display_ph */
```

```
void display_world(W)
    WORLD *W;
{
  POLYHEDRON *PH;
  POLYGON *PG;
  int NUM_PH=1;

if (W) {
  printf("\nWorld Name: %s",W->NAME);
  printf("\n\nWorld has:\n %d POLYHEDRONS\n ",W->DEGREE);
  PH=W->POLYHEDRON_LIST;
  while (PH) {
    printf("\n\nPH #%d \n",NUM_PH);
    NUM_PH++;
    display_ph(PH);
    PH=PH->NEXT;
  }
}  /* end if */
}  /* end display world */

/*********************  ***************************************

   The following functions are used by the construction file
   to add structures (i.e.- POLYHEDRON, POLYGON, VERTEX, and INSTANCE)
   and associations (i.e.- vertical edges and floor->ceiling associations)
   to a world.

*********************************************************************/

void add_edge(V1,V2)
    VERTEX *V1, *V2;  /*lower and upper vertices of edge*/
{
  if (V1->VERT_EDGE)
    printf("\nWarning reassignment of vertical edge attempted!!!");
  else
    V1->VERT_EDGE = V2;
} /* end add_edge */


/*********************************************************************/

void add_ceiling(PG,C)
    POLYGON *PG, *C;    /*floor and its new ceiling*/
{
  POLY_LINK *NEW_C,*NEXT_C;
  int FOUND=0;

  if (PG->CEILING_LIST) {
    NEXT_C= PG->CEILING_LIST;
    if (NEXT_C->REF_POLY==C)
      FOUND=1;
    else
      while (NEXT_C->NEXT) {
            if (NEXT_C->NEXT->REF_POLY==C)
              FOUND=1;
            NEXT_C=NEXT_C->NEXT;
      } /* end while */
    if (FOUND==0) {
      NEW_C=create_poly_link();  /*link onto end of list*/
      NEW_C->REF_POLY=C;
      NEW_C->PREV=NEXT_C;
      NEXT_C->NEXT=NEW_C;
```

```
    PG->C_DEGREE++;
   } /* end if */
   else
    printf("\nWarning - attempted to add ceiling which exists");
  } /* end if */
  else {
   NEW_C=create_poly_link(); /*adding 1st ceiling to list*/
   NEW_C->REF_POLY=C;
   PG->CEILING_LIST=NEW_C;
   PG->C_DEGREE++;
  } /* end else */
} /* end add_ceiling */

/*****************************************************************
 X,Y,Z is the position in the parent world at which the pivot point
 is to be placed.
 PIVOT_X and PIVOT_Y specify th local coordinates (in POLYHEDRON) of
 the objects pivot point.
 ROT is the number of degrees the object should be rotated about this
 pivot point.
*****************************************************************/
void *add_instance(NAME,LEN,PH,X,Y,Z,PIVOT_X,PIVOT_Y,ROT)
    POLYHEDRON *PH;
    float X,Y,Z,PIVOT_X,PIVOT_Y,ROT;
    char NAME[];    /*label for instance and number of characters in label*/
    int LEN;
{
 INSTANCE *I,*TEMP_I,*NEXT_I;
 int i;
 I=create_instance();  /*allocate and initialize memory*/
 for (i=0;i<=LEN;++i) {
  I->NAME[i]=NAME[i];
 }
 I->X=X;
 I->Y=Y;
 I->Z=Z;
 I->PIVOT_X=PIVOT_X;
 I->PIVOT_Y=PIVOT_Y;
 I->ROTATION=ROT;
 /*order by z*/
 if (PH->INSTANCE_LIST==NULL) {
  PH->INSTANCE_LIST=I;
 }
 else {
  NEXT_I=PH->INSTANCE_LIST;
  if (Z<=NEXT_I->Z) {
   I->NEXT=NEXT_I;
   NEXT_I->PREV=I;    /* add to head of list*/
   PH->INSTANCE_LIST=I;
  } /* end if */
  else {
   while (NEXT_I->NEXT&&NEXT_I->NEXT->Z<Z) {
        NEXT_I=NEXT_I->NEXT;  /*scan to insertion point*/
    }
   if (NEXT_I->NEXT) {
        I->NEXT=NEXT_I->NEXT; /*add to middle of list*/
        I->PREV=NEXT_I;
        NEXT_I->NEXT=I;
        I->NEXT->PREV=I;
   } /* end if */
   else {
        I->PREV=NEXT_I;    /*add as last instance*/
```

```
            NEXT_I->NEXT=I;
        } /* end else */
      } /* end else */
    } /* end else */
    PH->I_DEGREE++;          /*keep track of the number of instances*/
} /* end add_instance */

/*********************************************************************

  The remaining add functions create and add structures to the world.
  Pointers to each newly added structure are returned to the caller for
  future use.

*********************************************************************/


VERTEX *add_vertex(PG,X,Y)
     POLYGON *PG;    /* parent polygon to add vertex to*/
     float X,Y;      /*local coordinates of vertex*/
{
  VERTEX *V, *NEXT_V;

  V=create_vertex();
  V->X=X;
  V->Y=Y;
  if (PG->VERTEX_LIST==NULL)
    PG->VERTEX_LIST=V;
  else {
    NEXT_V=PG->VERTEX_LIST;
    while (NEXT_V->NEXT) {
      NEXT_V=NEXT_V->NEXT;     /* scan to end of list */
    }
    NEXT_V->NEXT=V;   /* add to end of list to retain order added*/
    V->PREV=NEXT_V;
  } /* end else */
  PG->DEGREE++;
  return V;
} /* end add_vertex */

/*-------------------------------------------------------------*/

POLYGON *add_pg(PH,Z,FLOOR,CONVEX)
     POLYHEDRON *PH;          /*parent structure*/
     float Z;                 /*height in local coordinates*/
     int FLOOR,CONVEX;        /*boolean values*/
{
     POLYGON *PG,*NEXT_PG;

  PG=create_polygon();
  PG->Z_VALUE=Z;
  PG->FLOOR=FLOOR;
  PG->CONVEX=CONVEX;
  if (PH->POLYGON_LIST==NULL)   /*sorted by Z height*/
    PH->POLYGON_LIST=PG;
  else {
    NEXT_PG=PH->POLYGON_LIST;
    if (Z<NEXT_PG->Z_VALUE) {   /*put at head of list*/
      NEXT_PG->PREV=PG;
      PG->NEXT=NEXT_PG;
      PH->POLYGON_LIST=PG;
    } /* end if */
    else {
```

```
    while ((NEXT_PG->NEXT)&&(NEXT_PG->NEXT->Z_VALUE>Z)){
        NEXT_PG=NEXT_PG->NEXT;
    }
    if (NEXT_PG->NEXT) {
        PG->NEXT=NEXT_PG->NEXT;   /* put in middle of list */
        PG->PREV=NEXT_PG;
        NEXT_PG->NEXT=PG;
        PG->NEXT->PREV=PG;
    } /* end if */
    else {
        NEXT_PG->NEXT=PG;   /* put at end of list */
        PG->PREV=NEXT_PG;
    } /* end else */
  } /* end else */
 } /* end else */
 PH->DEGREE++;
 return PG;
}   /* end add_pg */

/*-------------------------------------------------------------------*/

POLYHEDRON *add_ph(CLASS,LEN,W,FIXED,OBSTACLE)
    char CLASS[];     /*class name*/
    WORLD *W;         /*world to add polyhedron to*/
    int FIXED,OBSTACLE,LEN;   /* 2 booleans and the length of CLASS*/
{
 POLYHEDRON *PH,*NEXT_PH;
 int i;

 PH=create_polyhedron();
 for (i=0;i<=LEN;++i) {
  PH->CLASS[i]=CLASS[i];
 }
 PH->FIXED=FIXED;
 PH->OBSTACLE=OBSTACLE;
 if (W->POLYHEDRON_LIST==NULL)
  W->POLYHEDRON_LIST=PH;
 else {
  NEXT_PH=W->POLYHEDRON_LIST;
  while (NEXT_PH->NEXT) {
   NEXT_PH=NEXT_PH->NEXT;    /*scan to end of list*/
  }
  NEXT_PH->NEXT=PH;
  PH->PREV=NEXT_PH;
 } /* end else */
 W->DEGREE++;
 return PH;
}  /* end add_ph */

/*-------------------------------------------------------------------*/
WORLD *add_world(NAME,LEN)
    char NAME[];  /*label and its length*/
    int LEN;
{
 WORLD *W;
 int i;
 W=create_world();
 for (i=0;i<LEN;++i) {  /*assign label*/
  W->NAME[i]=NAME[i];
 }
 return W;
} /* end add_world */
```

```
/*************************************************************************
  find_ph will find and display a polyhedron based on its class
  name. Component polygons and instances will be listed to the screen.
  If the pointer to a polyhedron is needed: change this function
  return PH
*************************************************************************/

void find_ph(LABEL,W)
     char LABEL[MAX_LEN];     /*class label to look for*/
     WORLD *W;                /*world to search*/
{
  POLYHEDRON *NEXT_PH, *PH;
  int FOUND=0, i, MATCH;

  if (W) {
    printf("\nsearching for label: (");
    for (i=0;i<MAX_LEN;++i) {
            printf("%c",LABEL[i]);
    }
    printf(")\n");
    NEXT_PH=W->POLYHEDRON_LIST;
    while (NEXT_PH) {
      MATCH=1;
      for (i=0;i<MAX_LEN;++i) {
            if (NEXT_PH->CLASS[i]!=LABEL[i]){
              MATCH=0;        /*at least one character is different*/
            }
      }
      if (MATCH==1) {
            FOUND++;
            PH=NEXT_PH;
      }
      NEXT_PH=NEXT_PH->NEXT;
    } /* end while */
    if (FOUND==0)
      printf("\nNo polyhedron found under this label!\n");
    else {
      display_ph(PH);    /*show the polygon found*/
      if (FOUND>1)
            printf("\nWarning non-unique label (last occurance listed).\n");
    } /* end else */
  } /* end if */
  else
    printf("\n\nCannot find polyhedron since world is empty !!!\n");
} /* end find_ph */
```

```
/*
FILE NAME:    visibility.h
AUTHOR:       James Stein
PROJECT:      Thesis, supporting Yamabico vision system
DATE:         March 1992
ADVISOR:      Dr. Kanayama

COMMENTS: This file implements a algorithm which determines the set of
visible line seen from a given position in a wire frame model. The observer is
assumed to have omni-directional vision. To impose the physical limits of
a camera's field of view, the function get_view in file graphics.h can be
sent a model (as it in turn uses this file).

Primary Function(s):

      - conduct_vsiibility_sweep

          INPUT:    W                    a pointer to a 2d+ world model
                    EYE_X,EYE_Y,EYE_Z    position of observer in model W

          OUTPUT:   LINE_LIST            structure pointing to 2 lists of
                                         visible vertical and non-vertical lines
*/



/*---------------------- Structure definitions:----------------------------*/


typedef struct sweep_link {
          double THETA, X, Y, Z,
               MIN_Z, MAX_Z, UPPER_Z, DIST;
          VERTEX *V;
          INSTANCE *I;
          POLY_LINK *CEILINGS;
          struct sweep_link *PREV, *NEXT;
} SWEEP_LINK;


/*-------------------------------------------------------------------*/


typedef struct considered_link {
          double MIN_SWEEP,
                    MIN_Z, MAX_Z, DIST,
                    NEW_MIN_Z,NEW_MAX_Z,UPPER_Z;
          int    VISIBILITY,B_VISIBILITY,NEW_VISIBILITY,NEW_B_VISIBILITY;
          POLY_LINK *CEILINGS;
          SWEEP_LINK *SL1, *SL2;
          struct considered_link *NEXT;
} CONSIDERED_LINK;
```

```
/*-----------------------------------------------------------------*/


typedef struct considered_head {
        CONSIDERED_LINK *LINKS;
} CONSIDERED_HEAD;

/*-----------------------------------------------------------------*/


/* global variables: */

static double X,Y,Z;  /*Position of observer within the model*/
static THETA;         /*Current angle of visibility sweep*/
int IN_MAIN;          /*if 0 we are still preprocessing straddlers*/



void line_ray_intersection(CONSIDERED_LINK *CL,double ANGLE,
                           double *INT_X,double *INT_Y,double *DIST);


/* Doubles can be truncated to 4 decimal places to compensate for inexactness
   of floating point operations*/

double trunc(X)
    double X;
{
  int DUMMY;
  double XX=X;

  DUMMY=XX*10000;
  XX=DUMMY;
  XX=XX/10000.0;
  return XX;
}

/*****************************CONVERSIONFUNCTIONS*********************/

double degs(RADS)
    double RADS;
{
  return trunc(RADS*180.0/PI);
}



double rads(DEGS)
    double DEGS;
{
  return trunc(DEGS*PI/180.0);
}
```

```
/*************************************************************************/
/* Determin  if the edges from 2 considered links are colinear*/

int colinear(F,B)
        CONSIDERED_LINK *F,*B;
{
        double M1,M2;  /*we will compare slopes and distance*/

        M1=trunc((F->SL1->Y-F->SL2->Y)/(F->SL1->X-F->SL2->X));
        M2=trunc((B->SL1->Y-B->SL2->Y)/(B->SL1->X-B->SL2->X));
        if ((M1==M2)&&(F->DIST==B->DIST))
                return 1;
        else
                return 0;
}


/*****************************COUNTERCLOCKWISE CHECKS*********************/

int ccw(SL,PREV_SL)
        SWEEP_LINK *SL, *PREV_SL;
{
        double AREA;

        AREA= 0.5*((SL->X-X)*(PREV_SL->Y-Y)-
                (PREV_SL->X-X)*(SL->Y-Y));
        if (AREA>0.0)
          return 1;
        else
          return 0;
}  /* end ccw */



int ccw2(SL1,SL2,SL3)
        SWEEP_LINK *SL1,*SL2,*SL3;
{
        double AREA;

        AREA= 0.5*((SL2->X-SL1->X)*(SL3->Y-SL1->Y)-
                (SL3->X-SL1->X)*(SL2->Y-SL1->Y));
        if (AREA>0.0)
          return 1;
        else
                return 0;
}  /* end ccw */

/*************************************************************************/
```

```
/* Finds the angle from X1,Y1 to V for use in determining if X1,Y1 lies
   within the bounds of a polygon.*/

double find_theta(X1,Y1,V,I)
        double X1,Y1;
        VERTEX *V;
        INSTANCE *I;
{
        double X2,Y2,T;
        double LOCAL_X,LOCAL_Y,ROT_X,ROT_Y,RADS;

        LOCAL_X = V->X - I->PIVOT_X;
        LOCAL_Y = V->Y - I->PIVOT_Y;

 /* rotate about the z axis */
        RADS = I->ROTATION * PI / 180.0 ;   /* convert degs to rads */
        ROT_X = (cos(RADS)*LOCAL_X)+(sin(RADS)*LOCAL_Y);
        ROT_Y = (cos(RADS)*LOCAL_Y)-(sin(RADS)*LOCAL_X);

  /* translate to proper position in world model */

        X2 = I->X + ROT_X;
        Y2 = I->Y + ROT_Y;
        if ((X1==X2)||((Y1==Y2)&&(X1==X2)))
                T=0.0;
        else
                T=atan2(Y2-Y1,X2-X1);  /* both won't be 0 */
        if (T<0.0)
                T+=rads(360.0); /* normalize to 0-360 */
        return T;
}  /* end find_theta */



/* This function determines if the point X1,Y1 lies within the polygon, PG.
   The angle formed between lines drawn to each edge of PG is calculated.
   CW angles are added and CCW ones are subtracted from the sum.
   If the sum is not equal to 0.0 the point is within PG and 1 is returned.*/

int in_polygon(X1,Y1,PG,I)
        double X1,Y1;
        POLYGON *PG;
        INSTANCE *I;
{
        VERTEX *FIRST_V, *V=PG->VERTEX_LIST;
        double THETA1,THETA2,FIRST_THETA,SUM=0.0,SUM1=0.0;
        double XX,YY;


        THETA2=find_theta(X1,Y1,V,I);
        FIRST_V=V;
        FIRST_THETA=THETA2;
        while (V->NEXT) {
                if ((X1==V->X)&&(Y1==V->Y))
                        SUM1=1.0;       /*if directly under a point accept*/
                THETA1=THETA2;
                THETA2=find_theta(X1,Y1,V->NEXT,I);
/*ccw*/
                if ((0.5*((V->X-X1)*(V->NEXT->Y-Y1)-
                   (V->NEXT->X-X1)*(V->Y-Y1)))>0.0) {
                        if (THETA2<THETA1)
                                SUM+=(THETA2+rads(360.0))-THETA1;
```

```
                        else
                                SUM+=THETA2-THETA1;
                }
/*cw*/          else {
                        if (THETA2>THETA1)
                                SUM+=THETA2-(THETA1+rads(360.0));
                        else
                                SUM+=THETA2-THETA1;
                }
                V=V->NEXT;
        } /* end while */
/*Lastly: check the closing edge to see if we add or subtract its angle*/
        THETA1=THETA2;
        THETA2=FIRST_THETA;
        if ((0.5*((V->X-X1)*(FIRST_V->Y-Y1)-(FIRST_V->X-X1)*(V->Y-Y1)))>
           0.0) {  /*ccw*/
                if (THETA2<THETA1)
                        SUM+=(THETA2+rads(360.0))-THETA1;
                else
                        SUM+=THETA2-THETA1;
        }
        else {
                if (THETA2>THETA1)
                        SUM+=THETA2-(THETA1+rads(360.0));
                else
                        SUM+=THETA2-THETA1;
        }
        if (((trunc(SUM)==0.0))&&(SUM1==0.0))
                return 0;
        else
                return 1;
}


/* Function checks to see which ceiling of CL's ceiling list the 1st endpoint
   falls under. This height is returned and is used to determine how much
   coverage the CL has along the z-axis (that is what angle hound the portion
   of the z-axis which CL occludes*/

double find_ceiling_z(CL)
        CONSIDERED_LINK *CL;
{
        double IX,IY,DIST,CEILING_Z_VALUE=(-9999999.9);
        POLY_LINK *NEXT_C=CL->CEILINGS;
        int FOUND=0;

        IX=CL->SL1->X;
        IY=CL->SL1->Y;
        while (NEXT_C) {
/*keep track of highest ceiling over CL*/
        if ((in_polygon(IX,IY,NEXT_C->REF_POLY,CL->SL1->I)==1)&&
           (NEXT_C->REF_POLY->Z_VALUE+CL->SL1->I->Z>CEILING_Z_VALUE)){
                CEILING_Z_VALUE=NEXT_C->REF_POLY->Z_VALUE+CL->SL1->I->Z;
                FOUND=1;
        }
          NEXT_C=NEXT_C->NEXT;
        }
        if (FOUND==0) {
          CEILING_Z_VALUE=CL->SL1->UPPER_Z; /*if none ht same as endpoint*/
        }
        return trunc(CEILING_Z_VALUE);      /*return highest ceiling ht*/
} /* end find_ceiling_z */
```

```
/* Calculate the minimum and maximum angles which SL covers on the z-axis.
   Any object which is farther away and behind SL that falls within these
   limits will not be able to be seen. */

void calc_z_coverage(SL)
        SWEEP_LINK *SL;
{
        double dz,LEN;

        dz=SL->Z-Z;
        LEN=SL->DIST;  /*dist to line in X-Y plane*/
        if (LEN==0)
                LEN=0.00001;
        SL->MIN_Z=trunc(atan(dz/LEN));
        dz=SL->UPPER_Z-Z;
        SL->MAX_Z=trunc(atan(dz/LEN));
}  /* end calc_z_coverage */


/*  Absolute value of a double */

double my_abs(A)
        double A;
{
        if (A>=0.0)
                return A;
        else
                return -A;
}


/* Calculates the limiting angles along the z-axis for each item on the
   considered list. These limits are based upon the height of each
   endpoint (value of CL->MIN_Z) and the height of the ceiling (if any)
   lying above CL (CL->UPPER_Z). */

void calc_current_z_coverage(CLIST)
        CONSIDERED_HEAD *CLIST;
{
        CONSIDERED_LINK *CL=CLIST->LINKS;
        double MIN,MAX,DIST;
        double dx,dy,dz,IX,IY,LEN1,LEN2,
                CEILING_Z;

  while (CL) {
        CL->NEW_MAX_Z=trunc(atan((CL->UPPER_Z-Z)/CL->DIST));
        CL->NEW_MIN_Z=trunc(atan((CL->SL1->Z-Z)/CL->DIST));
        CL->NEW_VISIBILITY=1;    /*reset visibilities*/
        CL->NEW_B_VISIBILITY=1;
        CL=CL->NEXT;
  }
}  /* end calc_current_z_coverage */
```

```
/**********MEMORYALLOCATION FUNCTIONS**********/

LINE_HEAD *make_line_head()
{
        LINE_HEAD *LH=(LINE_HEAD *)malloc(sizeof(LINE_HEAD));

        LH->LINES=0;
        LH->VERT_LINES=0;
        LH->LINE_LIST=NULL;
        LH->TAIL=NULL;
        LH->VLINE_LIST=NULL;
        LH->VTAIL=NULL;
        return LH;
}  /* end make_line_head */


CONSIDERED_HEAD *make_considered_head()
{
        CONSIDERED_HEAD *CH;

        CH= (CONSIDERED_HEAD *)malloc(sizeof(CONSIDERED_HEAD));
        CH->LINKS=NULL;
        return CH;
}  /* end make_considered_head */
```

```
SWEEP_LINK *make_sweep_link(PH,PG,V,I,PG_Z)
        POLYHEDRON *PH;
        POLYGON *PG;
        VERTEX *V;
        INSTANCE *I;
        double PG_Z;
{
        SWEEP_LINK *SL;
        double LOCAL_X,LOCAL_Y,ROT_X,ROT_Y,RADS;

        SL= (SWEEP_LINK *)malloc(sizeof(SWEEP_LINK));
        SL->PREV= NULL;
        SL->NEXT= NULL;
        SL->V=V;
        SL->I=I;
        SL->CEILINGS=PG->CEILING_LIST;
        LOCAL_X = V->X - I->PIVOT_X;
        LOCAL_Y = V->Y - I->PIVOT_Y;

 /* rotate about the z axis */
        RADS = I->ROTATION * PI / 180.0 ;   /* convert degs to rads */
        ROT_X = (cos(RADS)*LOCAL_X)+(sin(RADS)*LOCAL_Y);
        ROT_Y = (cos(RADS)*LOCAL_Y)-(sin(RADS)*LOCAL_X);

  /* translate to proper position in world model */

        SL->X = trunc(I->X + ROT_X);  /*must be truncated*/
        SL->Y = trunc(I->Y + ROT_Y);
        SL->Z = trunc(I->Z + PG_Z);

        SL->THETA=(atan2(SL->Y-Y,SL->X-X));  /* both won't be 0 */
        if (SL->THETA<0.0)
                SL->THETA=(2.0*PI+SL->THETA); /* normalize to 0-360 */
        SL->DIST= trunc(sqrt(pow((SL->Y-Y),2.0)+pow((SL->X-X),2.0)));
        if (V->VERT_EDGE)
                SL->UPPER_Z=find_z(PH,V->VERT_EDGE)+I->Z;
        else
                SL->UPPER_Z=SL->Z;
        if ((PH->OBSTACLE==0)&&(PG->FLOOR==0))
                SL->UPPER_Z=99999999999.9;/*max float to cover 90 degs*/
        calc_z_coverage(SL);
        return SL;
}  /* end make_sweep_link */
```

```
CONSIDERED_LINK *make_considered_link(SL)
        SWEEP_LINK *SL;
{
        CONSIDERED_LINK *CL=(CONSIDERED_LINK *)malloc(sizeof(CONSIDERED_LINK));

        CL->SL1=SL;
        CL->DIST=SL->DIST;
        CL->SL2=SL->PREV;
        CL->CEILINGS=SL->CEILINGS;
        CL->VISIBILITY=1;
        CL->B_VISIBILITY=1;
        CL->NEW_VISIBILITY=1;
        CL->NEW_B_VISIBILITY=1;
        CL->NEXT=NULL;
        CL->MIN_SWEEP=SL->THETA;/*set  min to reflect sweep so far*/
        CL->MIN_Z=SL->MIN_Z;
        CL->MAX_Z=SL->MAX_Z;
        if (CL->SL1->UPPER_Z>9999999.9)  /*need to trunc????*/
                CL->UPPER_Z=99999999999.9;
        else
                CL->UPPER_Z=find_ceiling_z(CL);
        return CL;
} /* make_considered_link */

/*************************************************************************/



/***************************MEMORYDEALLOCATION*****************************/

void    free_sweep_list(SLIST)
        SWEEP_LINK *SLIST;
{
        SWEEP_LINK *TRASH=SLIST;

        while (TRASH) {
                SLIST=SLIST->NEXT;
                free(TRASH);
                TRASH=SLIST;
        }
} /* end free_sweep_list */


void free_clist(CLIST)
        CONSIDERED_HEAD *CLIST;
{
        CONSIDERED_LINK *NEXT_CL=CLIST->LINKS,*TRASH;

        while (NEXT_CL) {
                TRASH=NEXT_CL;
                NEXT_CL=NEXT_CL->NEXT;
                free(TRASH);
        }
        free(CLIST);
}  /* end free_clist */

/*************************************************************************/
```

```
/************************DISPLAYFUNCTIONS*******************************
 NOTE: These functions were used in debugging, but they have been left
       in case inspection of intermediate results is needed in the future.*/


void print_l(L)
        LINE *L;
{
        printf("\n\nline: X1 = %.2lf Y1 = %.2lf Z1 = %.2lf ",L->X1,L->Y1,L->Z1);
        printf("\n      X2 = %.2lf Y2 = %.2lf Z2 = %.2lf \n",L->X2,L->Y2,L->Z2);
        fflush(stdout);
}


void print_llist(LIST)
        LINE_HEAD *LIST;
{
        LINE *NEXT_L=LIST->VLINE_LIST;

        printf("\n\n\nVertical lines (%d) are:\n\n",LIST->VERT_LINES);
        while (NEXT_L) {
                print_l(NEXT_L);
                NEXT_L=NEXT_L->NEXT;
        }
        printf("\n\n\nnon-vertical lines (%d) are:\n\n",LIST->LINES);
        fflush(stdout);
        NEXT_L=LIST->LINE_LIST;
        while (NEXT_L) {
                print_l(NEXT_L);
                NEXT_L=NEXT_L->NEXT;
        }
}

void print_sl(SL)
        SWEEP_LINK *SL;
{
        printf("\nSL:  X = %.2lf Y = %.2lf Z = %.2lf",SL->X,SL->Y,SL->Z);
        printf("\n     THETA = %.20lf DIST = %.2lf",degs(SL->THETA),SL->DIST);
        printf("\n     MIN_Z = %.2lf MAX_Z = %.2lf",
                degs(SL->MIN_Z),degs(SL->MAX_Z));
        if (SL->PREV==NULL)
                printf("\nWarning no previous link");
        if (SL->NEXT==NULL)
                printf("Warning should be last link");
        fflush(stdout);

} /* end print_sl*/


void print_slist(SL)
        SWEEP_LINK *SL;
{
        SWEEP_LINK *NEXT_SL=SL;

        printf("\n\nSWEEP LIST:\n\n");
        while (NEXT_SL) {
                print_sl(NEXT_SL);
                NEXT_SL=NEXT_SL->NEXT;
        }
}
```

```
void print_cl(CL)
        CONSIDERED_LINK *CL;
{

        printf("\n\n     MIN_Z=%.2lf MAX_Z=%.2lf",
             degs(CL->MIN_Z),degs(CL->MAX_Z));
        printf("\n\n NEW_MIN_Z=%.2lf NEW_MAX_Z=%.2lf",
             degs(CL->NEW_MIN_Z),degs(CL->NEW_MAX_Z));
        printf("\n     MIN_SWEEP=%.2lf  DIST=%.2lf",
                degs(CL->MIN_SWEEP),CL->DIST);
        printf("\nUPPER_Z:%.2lf",CL->UPPER_Z);
        printf("\nOLD: VISIBLE=%d  B_VISIBLE=%d",CL->VISIBILITY,CL->B_VISIBILITY);
        printf("\nNEW: VISIBLE=%d  B_VISIBLE=%d",
                CL->NEW_VISIBILITY,CL->NEW_B_VISIBILITY);
        print_sl(CL->SL1);
        print_sl(CL->SL2);
}  /* end print_cl */


void print_clist(CLIST)
        CONSIDERED_HEAD *CLIST;
{
        CONSIDERED_LINK *CL=CLIST->LINKS;

        printf("\n\nConsidered list (THETA=%.2lf):\n\n",degs(THETA));
        while (CL) {
                print_cl(CL);
                CL=CL->NEXT;
        }
} /* end print_clist */

/*************************************************************************/


/* Sweep links are added to the list in order of their THETA values*/

SWEEP_LINK *add_sweep_link(LIST,LINK)
        SWEEP_LINK *LIST, *LINK;
{
        SWEEP_LINK *TEMP;

        if (LIST) {
          TEMP=LIST;
          if (TEMP->THETA>LINK->THETA) {
                LINK->NEXT=LIST;
                LIST=LINK;   /* inserted as 1st element */
          }
          else {
                while ((TEMP->NEXT)&&(TEMP->NEXT->THETA<=LINK->THETA)) {
                        TEMP=TEMP->NEXT;
                }
                LINK->NEXT=TEMP->NEXT;
                TEMP->NEXT=LINK;
          } /* end else */
        } /* end if */
        else
          LIST=LINK;   /* is first element to add to list */
        return LIST;
}  /* end add_sweep_link */
```

```
/*  This function scans through the entire world model (W). A sweep link is
    made for each vertex of the model. The angle from the observer (global
    variable) to the vertex is calculated and used to sort the links.
    When a link is made, we also inspect its ->VERT_EDGE pointer to see
    if a vertical line leaves it. Calculate_z_coverage uses the height of
    this vertical line to determine coverage of the vertex along the z-axis.
      Each sweep link has its PREV pointer assigned to indicate the link
    which preceeded it in the polygon list. In latter processing only sweep
    links with a ccw relationship to this PREV link will be considered as
    visible.
      Since we will latter require all floors residing above the observer and
    all ceiling below them to be visible, we inspect each polygon for these
    properties. When a polygon satifies one of these, it's vertices are
    processed a second time in reverse order. This ensures that every edge
    of the polygon will swho up as a ccw CONSIDERED_LINK.
    */

SWEEP_LINK *make_sweep_list(W)
        WORLD *W;
{
        SWEEP_LINK *SWEEP_LIST=NULL,*NEXT_L,*LAST_L,*FIRST_L;
        POLYHEDRON *NEXT_PH;
        POLYGON *NEXT_PG;
        VERTEX *NEXT_V, *LAST_V;
        INSTANCE *NEXT_I, *LAST_I;

    NEXT_PH=W->POLYHEDRON_LIST;
    while (NEXT_PH) {
      NEXT_I=NEXT_PH->INSTANCE_LIST;
      while (NEXT_I) {
        NEXT_PG=NEXT_PH->POLYGON_LIST;
        while (NEXT_PG) {
          NEXT_V=NEXT_PG->VERTEX_LIST;
          NEXT_L=make_sweep_link(NEXT_PH,NEXT_PG,NEXT_V,NEXT_I,
                                 NEXT_PG->Z_VALUE);
          SWEEP_LIST=add_sweep_link(SWEEP_LIST,NEXT_L);/*make and add links*/
          FIRST_L=NEXT_L;
          LAST_L=NEXT_L;
          NEXT_V=NEXT_V->NEXT;
          while (NEXT_V) {
              NEXT_L=make_sweep_link(NEXT_PH,NEXT_PG,NEXT_V,NEXT_I,
                                     NEXT_PG->Z_VALUE);
              NEXT_L->PREV=LAST_L;
              SWEEP_LIST=add_sweep_link(SWEEP_LIST,NEXT_L);
              LAST_L=NEXT_L;
              NEXT_V=NEXT_V->NEXT;
             } /* end while */
          FIRST_L->PREV=LAST_L;   /* add line which closes polygon */
/* Make entire polygon ccw so it may be visible */
          if ((((NEXT_PG->Z_VALUE+NEXT_I->Z<Z)&&(NEXT_PG->FLOOR==0))||
               ((NEXT_PG->Z_VALUE+NEXT_I->Z>Z)&&(NEXT_PG->FLOOR==1)))||
               ((NEXT_PH->OBSTACLE==0)&&(NEXT_PG->FLOOR==0))){
```

```
/* To cut down on processing time the above if statement can be commented
out and the below one used. This has the effect of assuming a model
is composed of only large objects (observer doesn't look down or up to them).
We still must make enclosure ceiling visible since items such as door jam
ceilings will not always be above the observe*/

/*          if ((NEXT_PH->OBSTACLE==0)&&(NEXT_PG->FLOOR==0)){*/
              NEXT_V=NEXT_PG->VERTEX_LIST;
              NEXT_L=make_sweep_link(NEXT_PH,NEXT_PG,NEXT_V,NEXT_I,
                                     NEXT_PG->Z_VALUE);
            if (!((NEXT_PH->OBSTACLE==0)&&(NEXT_PG->FLOOR==0))){
              NEXT_L->MAX_Z=NEXT_L->MIN_Z; /*take away height if any*/
              NEXT_L->CEILINGS==NULL;
              NEXT_L->UPPER_Z=NEXT_L->Z;
              }
              FIRST_L=NEXT_L;
              NEXT_V=NEXT_V->NEXT;
              while (NEXT_V) {
                    LAST_L=make_sweep_link(NEXT_PH,NEXT_PG,NEXT_V,NEXT_I,
                                           NEXT_PG->Z_VALUE);
            if (!((NEXT_PH->OBSTACLE==0)&&(NEXT_PG->FLOOR==0))){
                    LAST_L->MAX_Z=LAST_L->MIN_Z; /*take away height if any*/
                    LAST_L->CEILINGS==NULL;
                    LAST_L->UPPER_Z=LAST_L->Z;
              }
                    NEXT_L->PREV=LAST_L;
                    SWEEP_LIST=add_sweep_link(SWEEP_LIST,NEXT_L);
                    NEXT_L=LAST_L;
                    NEXT_V=NEXT_V->NEXT;
              }
              NEXT_L->PREV=FIRST_L;
              SWEEP_LIST=add_sweep_link(SWEEP_LIST,NEXT_L);
            }
            NEXT_PG=NEXT_PG->NEXT;
          } /* end while  NEXT_PG*/
          NEXT_I=NEXT_I->NEXT;
    } /* end while NEXT_I */
    NEXT_PH=NEXT_PH->NEXT;
  } /* end while NEXT_PH */
  return SWEEP_LIST;
}   /* end make_sweep_list */
```

```
/* Searches considered list (CL). if sweep link (SLINK) is the 2nd endpoint
   of an edge, that edge is returned to complete its processing. If no
   match is found a null pointer is returned*/

CONSIDERED_LINK *under_consideration(SLINK,CL)
        SWEEP_LINK *SLINK;
        CONSIDERED_HEAD *CL;
{
        CONSIDERED_LINK *NEXT_CL=CL->LINKS;

        while (NEXT_CL) {
        if (NEXT_CL->SL1==NULL)
                printf("\nWarning CL with no SL1");
        if (NEXT_CL->SL2==NULL)
                printf("\nWarning CL with no SL2");
                if (NEXT_CL->SL2==SLINK){
                        return NEXT_CL; /* retrun ptr if in list*/
                }
              else
                        NEXT_CL=NEXT_CL->NEXT;
        }
        return NEXT_CL;   /* returns NULL if not in list */
}  /* end under_consideration */



/* Determine the point of intersection along CL's edge which occurs with
   the ray originating from the observer's position (X,Y,Z) along ANGLE.
   The distance to this intersection ios also calculated.

   NOTE: Intersection and distance are returned by reference in variable
        addresses INT_X,INT_Y and DIST.
            It is assumed an intersection does take place (dictated by usage
            in algorithm).*/

void line_ray_intersection(CL,ANGLE,INT_X,INT_Y,DIST)
    CONSIDERED_LINK *CL;
    double ANGLE,*INT_X,*INT_Y,*DIST;
{
        double XX,YY;  /*values at intersection */
        double dx,dy;  /*delta values*/
        double M_LINE,M_RAY; /*slope of line and ray*/
        double B_LINE,B_RAY; /*y-interceepts*/

      dy=CL->SL2->Y-CL->SL1->Y;
        dx=CL->SL2->X-CL->SL1->X;
        if ((ANGLE==CL->SL1->THETA)&&(ANGLE==CL->SL2->THETA)){
          if (CL->SL1->DIST<=CL->SL2->DIST) {
                XX=CL->SL1->X;
                YY=CL->SL1->Y;
                *DIST=CL->SL1->DIST;
          }
          else {                                        /*colinear cases*/
                XX=CL->SL2->X;
                YY=CL->SL2->Y;
                *DIST=CL->SL2->DIST;
          }
        }
        else {
          if ((ANGLE==90.0)||(ANGLE==180.0)){  /*ray has no slope*/
                XX=X;
                M_LINE=dy/dx;
```

```
                YY=M_LINE*XX+(CL->SL1->Y-(M_LINE*CL->SLI->X));
        }
        else {
                M_RAY=tan(ANGLE);
                B_RAY=Y-M_RAY*X;
                if (CL->SL1->X==CL->SL2->X) { /*line has not slope */
                        XX=CL->SLI->X;
                        YY=M_RAY*XX+B_RAY;
                }
                else {                  /* both line and ray have a slope */
                  M_LINE=dy/dx;
                  B_LINE=CL->SLI->Y-M_LINE*CL->SLI->X;
                  XX=(B_LINE-B_RAY)/(M_RAY-M_LINE);
                  YY=M_RAY*XX+B_RAY;
                }  /* end else */
        }   /* end else */
        *DIST=trunc(sqrt(pow(XX-X,2.0)+pow(YY-Y,2.0)));  /*assign distance*/
      } /* end else */
      *INT_X=trunc(XX);    /*assign x-y coordinates of intersection*/
      *INT_Y=trunc(YY);
    } /* end line_ray_intersection */



/* Searches currently accepted lines. If L duplicates one of these, a 1 is
   returned. Duplications will naturally occure since each vertical line is
   common to 2 edges. */

int duplicate_vert_line(L,LIST)
        LINE *L;
        LINE_HEAD *LIST;
{
        int DUP=0;
        LINE *NEXT_L=LIST->VLINE_LIST;

        while (NEXT_L) {
                if ((L->X1==NEXT_L->X1)&&(L->Y1==NEXT_L->Y1)&&
                        (L->Z1==NEXT_L->Z1)&&(L->Z2==NEXT_L->Z2))
                        DUP=1;
                NEXT_L=NEXT_L->NEXT;
        }
        return DUP;
}  /* end duplicate_vert_line */



void add_vert_line(CL,SL,LIST)
        CONSIDERED_LINK *CL;
        SWEEP_LINK *SL;
        LINE_HEAD *LIST;
{
        LINE *NEW_LINE=(LINE *)malloc(sizeof(LINE));
        double len;

        len=SL->DIST;
        NEW_LINE->X1=SL->X;
        NEW_LINE->Y1=SL->Y;
        NEW_LINE->MODEL_X=SL->X;
        NEW_LINE->MODEL_Y=SL->Y;
        if (CL->MIN_Z>=SL->MIN_Z)
                NEW_LINE->Z1=tan(CL->MIN_Z)*len+Z;  /*clipped short*/
```

```
        else
                NEW_LINE->Z1=tan(SL->MIN_Z)*len+Z;
        NEW_LINE->X2=SL->X;
        NEW_LINE->Y2=SL->Y;
        if (CL->MAX_Z<=SL->MAX_Z)
                NEW_LINE->Z2=tan(CL->MAX_Z)*len+Z;   /*clipped short*/
        else
                NEW_LINE->Z2=tan(SL->MAX_Z)*len+Z;
        NEW_LINE->NEXT=NULL;
        if (duplicate_vert_line(NEW_LINE,LIST)==0){
                LIST->VERT_LINES++;
                if (LIST->VTAIL) {
                        LIST->VTAIL->NEXT=NEW_LINE; /*add as last vert. line*/
                        LIST->VTAIL=NEW_LINE;
                        }
                else {
                        LIST->VTAIL=NEW_LINE;  /* 1st vertical line added */
                        LIST->VLINE_LIST=NEW_LINE;
                }
        }  /* end if */
        else
                free(NEW_LINE);
}   /* end add_vert_line */


/* Adds only bottom edge of a considered link (CL). Lines are only accepted
   from their MIN_SWEEP angle to the current sweep angle (THETA).*/

void add_line(CL,LIST)
        CONSIDERED_LINK *CL;
        LINE_HEAD *LIST;
{
      LINE *NEW_LINE;
        double IX,IY,DIST;
                /*DIST req for call to intersection but value not used*/
 /*bottom line is visible and not just a single point*/
  if ((CL->B_VISIBILITY==1)&&(my_abs(CL->MIN_SWEEP-THETA)>0.0001)){
        NEW_LINE=(LINE *)malloc(sizeof(LINE));
        NEW_LINE->NEXT=NULL;
        LIST->LINES++;
        if (LIST->TAIL) {
                LIST->TAIL->NEXT=NEW_LINE; /* add non-vertical line*/
                LIST->TAIL=NEW_LINE;
                }
        else {
                LIST->TAIL=NEW_LINE;  /* 1st non-vertical line added */
                LIST->LINE_LIST=NEW_LINE;
        }
/* find first endpoint to accept*/
        line_ray_intersection(CL,CL->MIN_SWEEP,&IX,&IY,&DIST);
        NEW_LINE->X1=IX;
        NEW_LINE->Y1=IY;
        NEW_LINE->Z1=CL->SL1->Z;
/*find second endpoint*/
        line_ray_intersection(CL,THETA,&IX,&IY,&DIST);
        NEW_LINE->X2=IX;
        NEW_LINE->Y2=IY;
        NEW_LINE->Z2=CL->SL2->Z;
  }  /* end if */
  CL->MIN_SWEEP=THETA;
}  /* end add_line */
```

```
/* This function calculates distances from the observer along the current
   THETA to each edge on the considered list. Distances do not account for
   z infromation (height), but reflect straight line distance to the
   intersection lying in the x-y plane. */

void calculate_distances(CLIST)
        CONSIDERED_HEAD *CLIST;
{
        CONSIDERED_LINK *NEXT_CL=CLIST->LINKS;
        double IX,IY;  /*pointers and values at intersection */
        double DIST; /*distance to intersection values*/

  while (NEXT_CL) {
      line_ray_intersection(NEXT_CL,THETA,&IX,&IY,&DIST);
        NEXT_CL->DIST=DIST;
        NEXT_CL=NEXT_CL->NEXT;
  }  /* end while */
}  /* end calculate_distances */


/* When a link is put on the considered list, we must determine how much of
   it is blocked from view (along the z axis) and what affect it has on
   more distant edges.

   Notice that case 2 is not accounted for since we are dealing with a
   wire frame representation.*/

void calculate_visibility_add(CLINK,CLIST,LLIST)
        CONSIDERED_LINK *CLINK;
        CONSIDERED_HEAD *CLIST;
      LINE_HEAD *LLIST;
{
        CONSIDERED_LINK *CL=CLINK->NEXT;
        int TYPE_OCCLUSION;

        if (CLINK->NEW_VISIBILITY==1) {  /*if visible it may occlude others*/
          while (CL) {
                if (CL->NEW_VISIBILITY==1) { /*can only block visible lines*/
                        TYPE_OCCLUSION=occlusion(CLINK,CL);

                        switch (TYPE_OCCLUSION) {
                        case 4:                         /*totally occluded*/
                                CL->NEW_VISIBILITY=0;
                                CL->NEW_B_VISIBILITY=0;
                                break;
                        case 3:                 /*bottom occluded*/
                                CL->NEW_B_VISIBILITY=0;
                                CL->NEW_MIN_Z=CLINK->NEW_MAX_Z;
                                break;
/*                      case 2:
                                CL->NEW_MIN_Z=CLINK->NEW_MAX_Z;
                                break;
*/                      case 1:                         /*top occluded*/
                                CL->NEW_MAX_Z=CLINK->NEW_MIN_Z;
                                break;
                        } /*end switch*/
                  }  /* end if */
                  CL=CL->NEXT;
          }  /* end while */
        }  /* end if */
}  /* end calculate_visibility_add */
```

```
/* Calculate the visibility of the vertical edge (if any) residing on the
   2nd endpoint of a link which is being passed by the sweep (thus removed
   from the considered list)*/

void calc_vis_remove(CL,CLIST)
        CONSIDERED_LINK *CL;
        CONSIDERED_HEAD *CLIST;
{
        CONSIDERED_LINK *NEXT_CL=CLIST->LINKS;
        int TYPE_OCCLUSION;

/*now calc visibility bounds of SL2's vertical line if there is one*/
   if (CL->SL2->V->VERT_EDGE) {
        while ((CL!=NEXT_CL)&&(CL->NEW_VISIBILITY==1)){
                if (CL->SL2->THETA==NEXT_CL->SL1->THETA)
                   TYPE_OCCLUSION=0;
                else
                   TYPE_OCCLUSION=occlusion(NEXT_CL,CL);
                switch (TYPE_OCCLUSION) {
                  case 4:
                        CL->NEW_VISIBILITY=0;
                        break;
                  case 3: case 2:
                        CL->NEW_MIN_Z=NEXT_CL->NEW_MAX_Z;
                        break;
                  case 1:               /*top of B occluded*/
                        CL->NEW_MAX_Z=NEXT_CL->NEW_MIN_Z;
                        break;
                }  /*end switch*/
                NEXT_CL=NEXT_CL->NEXT;
        } /* end while */
      CL->VISIBILITY=CL->NEW_VISIBILITY;
        CL->MIN_Z=CL->NEW_MIN_Z;
        CL->MAX_Z=CL->NEW_MAX_Z;
   } /* end if */
   else
      CL->VISIBILITY=0;
}   /* end calc_vis_remove */

/* If visibility has been altered from last time, we must accept lines which
   were already visible and reset the value of MIN_SWEEP to reflect where along
   the edge these new values start.*/

int visibility_changes(CL)
        CONSIDERED_LINK *CL;
{
        int CHANGES=0;
        double EXP_MIN_Z, EXP_MAX_Z; /*expected coverage based on perspective*/

        EXP_MIN_Z=trunc(atan((CL->SL1->Z-Z)/CL->DIST));
        EXP_MAX_Z=trunc(atan((CL->UPPER_Z-Z)/CL->DIST));

        if (CL->VISIBILITY!=CL->NEW_VISIBILITY)
                CHANGES++;
        if (CL->B_VISIBILITY!=CL->NEW_B_VISIBILITY)
                CHANGES++;
        if (EXP_MIN_Z!=CL->NEW_MIN_Z)
                CHANGES++;
        if (EXP_MAX_Z!=CL->NEW_MAX_Z)
                CHANGES++;
        return CHANGES;
}
```

```
void update_visibility(CLIST,LLIST)
        CONSIDERED_HEAD *CLIST;
        LINE_HEAD *LLIST;
{
        CONSIDERED_LINK *CL=CLIST->LINKS;

        while (CL) {
                if (visibility_changes(CL)!=0) {
                  if ((CL->B_VISIBILITY==1)&&(IN_MAIN))
                        add_line(CL,LLIST);
                  CL->VISIBILITY=CL->NEW_VISIBILITY;
                  CL->B_VISIBILITY=CL->NEW_B_VISIBILITY;
                  CL->MIN_Z=CL->NEW_MIN_Z;
                  CL->MAX_Z=CL->NEW_MAX_Z;
                  CL->MIN_SWEEP=THETA;     /*values only affect here on*/
                }
                CL=CL->NEXT;
        }
}


/* Visibility must be periodically recomputed to account for the effects of
   perspective as the sweep progresses around 360 degrees.*/

void recompute_visibility(CLIST,LLIST)
        CONSIDERED_HEAD *CLIST;
        LINE_HEAD *LLIST;
{
        CONSIDERED_LINK *CL=CLIST->LINKS;

        calc_current_z_coverage(CLIST);    /*will change due to perspective*/
        while (CL) {                       /*add each link again*/
                calculate_visibility_add(CL,CLIST,LLIST);
                CL=CL->NEXT;
        }
        update_visibility(CLIST,LLIST);    /*see if changes occured*/
}  /* end recompute_visibility */

/* Add a new link to the considered list (sorted by distance from observer
   in the x-y plane). If a vertical edge resides on the links first endpoint
   accept it based on the edges computed visibility*/

void add_considered_link(CL,CLIST,LLIST)
        CONSIDERED_LINK *CL;
        CONSIDERED_HEAD *CLIST;
      LINE_HEAD *LLIST;
{
        CONSIDERED_LINK *NEXT_CL=CLIST->LINKS;

        if (CLIST->LINKS) {            /*recalc distances for insert*/
          calculate_distances(CLIST);
          if (CL->DIST<NEXT_CL->DIST) {
                CL->NEXT=CLIST->LINKS;
                CLIST->LINKS=CL;           /*add as 1st element*/
          }  /* end if */
          else {
                while ((NEXT_CL->NEXT)&&(NEXT_CL->NEXT->DIST<CL->DIST)){
                  NEXT_CL=NEXT_CL->NEXT;
              }
/*keep ones leaning in towards camera 1st on list*/
                while (((NEXT_CL->NEXT)&&(NEXT_CL->NEXT->DIST==CL->DIST))&&
                        (ccw2(CL->SL1,CL->SL2,NEXT_CL->NEXT->SL2))){
```

```
                        NEXT_CL=NEXT_CL->NEXT;
                    }
                    CL->NEXT=NEXT_CL->NEXT;
                    NEXT_CL->NEXT=CL;
                }   /* end else */
                recompute_visibility(CLIST,LLIST);
            }  /* end if */
            else {
                CLIST->LINKS=CL;          /*1st element added to null list*/
                CL->VISIBILITY=1;     /*so must be visible*/
                CL->B_VISIBILITY=1;     /*so must be visible*/
            }
            if ((IN_MAIN)&&(((CL->VISIBILITY==1)&&(CL->MIN_Z<CL->MAX_Z))
                &&(CL->SL1->V->VERT_EDGE)))
                add_vert_line(CL,CL->SL1,LLIST);
}   /* end add_considered_link */

/* Remove a CL from the list*/
void remove_cl(CL,CLIST)
            CONSIDERED_LINK *CL;
            CONSIDERED_HEAD *CLIST;
{
            CONSIDERED_LINK *NEXT_CL=CLIST->LINKS;

            if (CL==NEXT_CL) {  /* removing 1st link */
                CLIST->LINKS=NEXT_CL->NEXT;
                free(CL);    /*deallocate memory*/
            }
            else {
                while ((NEXT_CL->NEXT)&&(NEXT_CL->NEXT!=CL)) {
                    NEXT_CL=NEXT_CL->NEXT;
                }
                if (NEXT_CL->NEXT) {
                    NEXT_CL->NEXT=CL->NEXT;
                    free(CL);    /*deallocate memory*/
                }
            }  /* end else */
}   /* end remove_cl */

/* The sweep has progresses to the end of link CL. We need to inspect the
   visibility and accept both the bottom edge and vertical line (at 2nd
   endpoint) if required.
     Once this is done, visibility of the entire considered list (CLIST)
   must be recomputed to account for perspective and the deleted edge*/

void complete_line(CL,CLIST,LLIST)
            CONSIDERED_LINK *CL;
            CONSIDERED_HEAD *CLIST;
            LINE_HEAD *LLIST;
{
            LINE *L;

            if ((CL->VISIBILITY==1)&&(CL->B_VISIBILITY==1))
                add_line(CL,LLIST);   /*also checks for and adds right vert line*/
            calculate_distances(CLIST);
            calc_current_z_coverage(CLIST);
            calc_vis_remove(CL,CLIST);
            if ((CL->SL2->V->VERT_EDGE)&&(CL->VISIBILITY==1))
                add_vert_line(CL,CL->SL2,LLIST);
            remove_cl(CL,CLIST); /*if not visible no changes needed before removal*/
            recompute_visibility(CLIST,LLIST);
}   /* end complete_line */
```

```
/* These occlusion codes apply if both links begin at the same vertex*/
int overlay_occlusion(F,B)
        CONSIDERED_LINK *F, *B;
{
        int TYPE=0; /*default is no occlusion occurs*/
  if (F->NEW_MIN_Z<=B->NEW_MIN_Z) {
        if (F->NEW_MAX_Z>=B->NEW_MAX_Z)
                TYPE=4;    /* totally occluded*/
        else {
                if (F->NEW_MAX_Z>=B->NEW_MIN_Z)
                        TYPE=3;  /*bottom of B occluded*/
        }
  } /* end if */
  else {
        if (F->NEW_MAX_Z<B->NEW_MAX_Z)
                TYPE=2;    /*middle prtion of B occluded*/
        else {
                if (F->NEW_MIN_Z<=B->NEW_MAX_Z)
                        TYPE=1;  /*top of B occluded*/
        }
  } /* end else */
  /* otherwise there is no occlusion */
  return TYPE;
}   /* end overlay_occlusion */

/* The type of occlusion imposed upon the back edge (B) by the front edge (F)
   is determined: return value is 0,1,2,3, or 4 */
int occlusion(F,B)
        CONSIDERED_LINK *F, *B;
{
        int TYPE=0; /*default is no occlusion occurs*/

/*No occlusion if edges fall on the same plane or are end-to-end*/
if (((F->SL1->THETA==B->SL2->THETA)||(F->SL2->THETA==B->SL1->THETA))||
     ((F->MIN_Z==F->MAX_Z)||(colinear(F,B))))
   TYPE=0;
else {
   if (((F->SL1->DIST==B->SL1->DIST)&&(F->SL1->THETA==B->SL1->THETA))
        &&(B->UPPER_Z<9999.0))
        TYPE=overlay_occlusion(F,B);
   else {
     if (F->NEW_MIN_Z<B->NEW_MIN_Z) {
        if (F->NEW_MAX_Z>B->NEW_MAX_Z)
                TYPE=4;    /* totally occluded*/
        else {
                if (F->NEW_MAX_Z>B->NEW_MIN_Z)
                        TYPE=3;  /*bottom of B occluded*/
        }
     } /* end if */
     else {
        if (F->NEW_MAX_Z<B->NEW_MAX_Z)
                TYPE=2;    /*middle prtion of B occluded*/
        else {
                if (F->NEW_MIN_Z<B->NEW_MAX_Z)
                        TYPE=1;  /*top of B occluded*/
        }
      } /* end else */
   } /* end else */
 } /* end else */
  /* otherwise there is no occlusion */
  return TYPE;
}   /* end occlusion */
```

```
/* This is the primary function which will be called from outside this file.
A list of sweep links is constructed based on the model (W) and the observer's
position (EYE_X,EYE_Y,EYE_Z).
   Next edges straddling 0 degrees are placed on the considered list (if they
are ccw). Then main processing begins and each sweep link and its predicessor
pair is inspected. If the circuit from observer to SL to prev(SL) is ccw, the
SL's are put into a considered link (CL) and added to the considered list
(CLIST).
  As the sweep progresses throught the sweep links: visibility is updated,
lines are accepted, and edges are removed from CLIST (as they are passed).
    OUTPUT:     LINE_LIST    structure pointing to 2 list of lines
                      (vertical and non-vertical accepted lines)
*/
LINE_HEAD *conduct_visibility_sweep(W,EYE_X,EYE_Y,EYE_Z)
        WORLD *W;
        double EYE_X,EYE_Y,EYE_Z;
{
        SWEEP_LINK *NEXT_SL, *SWEEP_LIST=NULL;
        CONSIDERED_LINK *CL, *PAST_CL;
        CONSIDERED_HEAD *CLIST=make_considered_head();
        LINE_HEAD *LINE_LIST=make_line_head();
        int STRADDLERS=0;

        IN_MAIN=0;    /*still processing straddlers*/
        X=EYE_X;
        Y=EYE_Y;
        Z=EYE_Z;
        SWEEP_LIST=make_sweep_list(W);
        NEXT_SL=SWEEP_LIST;
/* Add all visible straddlers*/
        while (NEXT_SL) {
                THETA=NEXT_SL->THETA;
                if ((ccw(NEXT_SL,NEXT_SL->PREV)==1)&&
                   (NEXT_SL->THETA>NEXT_SL->PREV->THETA)) {
                        CL=make_considered_link(NEXT_SL);
                        add_considered_link(CL,CLIST,LINE_LIST);
                        CL->MIN_SWEEP=0.0;
                        STRADDLERS=1;
                }
                NEXT_SL=NEXT_SL->NEXT;
        } /* end while */
        NEXT_SL=SWEEP_LIST;
        THETA=0.0;
        IN_MAIN=1;
/* Process all of sweep list*/
        while (NEXT_SL) {
                THETA=NEXT_SL->THETA;
                while (PAST_CL=under_consideration(NEXT_SL,CLIST)){
                        complete_line(PAST_CL,CLIST,LINE_LIST);
                }
                if (ccw(NEXT_SL,NEXT_SL->PREV)==1) {
                        CL=make_considered_link(NEXT_SL);
                        add_considered_link(CL,CLIST,LINE_LIST);
                }
                NEXT_SL=NEXT_SL->NEXT;
        } /* end while */
```

```
        if (STRADDLERS) {   /* have lines crossing ZERO degrees */
                THETA=0.0;
                calculate_distances(CLIST);
                CL=CLIST->LINKS;
                while (CL) {
                        if ((CL->VISIBILITY==1)&&(CL->B_VISIBILITY==1))
                                add_line(CL,LINE_LIST);
                        CL=CL->NEXT;
                }
    } /* end if */
    free_clist(CLIST);
    free_sweep_list(SWEEP_LIST);
    return LINE_LIST;
} /* end conduct_visibility_sweep */
```

```
/*
FILE NAME:graphics.h
AUTHOR:   Lt James Stein
PROJECT:  Thesis, supporting Yamabico-11 vision system
Date:     March 1992
ADVISOR:  Dr. Kanayama

COMMENTS:
  This file contains the routines neccessary to support the projection of our
2d+ model world into a 2 dimensional window. This view will then be used for
pattern matching against the processed images extracted from the raw camera
data.

  Two primary functions are provide: get_view and get_full_view

-  get_view calls the function conduct_visibility_sweep in file "visibility.h"
the set of output lines represents a2d projection of all lines which should be
visible from a given position and orientation within the model world, W.

-  get_full_view does not call the visibility checking function. Its output
represents all model lines which are seen if everything in the model were
transparent

-  The memory deallocation function free_lines is provided also. The user can
send an uneeded LINE_HEAD pointer to this function for deallocation.


INPUT: Position in the model (PRPX,PRPY,PRPZ)
       orientation (ORIENT) with 0 degs being down the y-axis
       a world (W)
       focal length (FL)

The view angle of the camera is calculated based upon the camera's sensing
element size (constant CCD) and the supplied focal length (FL).
*/

/*CCD and clipping planes are in inches*/
#define CCD             (2.0/3.0)
#define NEARCLIP 1.0
#define FARCLIP 5000.0
#define MAX_X XMAXSCREEN   /*destination device (iris screen) limits*/
#define MAX_Y YMAXSCREEN

                        /*coordinates used by pattern matching*/
/*#define MAX_X 686.0
#define MAX_Y 486.0
*/


double VIEW_ANGLE;        /*width of camera's field of view in radians*/
```

```
/*-------------------------------------------------------------------*/
typedef struct line {
        double X1,X2,Y1,Y2,Z1,Z2; /*will hold final 2d device coordinates*/
        double MODEL_X,MODEL_Y;  /*original coordinates:line in the model*/
        int CLIP1[6],CLIP2[6];   /*clipping codes*/
        struct line *NEXT;
} LINE;


/*-------------------------------------------------------------------*/


typedef struct line_head {      /*vertical lines kept separate from others*/
        int LINES,VERT_LINES;
        LINE *LINE_LIST,*VLINE_LIST, *TAIL,*VTAIL;
} LINE_HEAD;


/*-------------------------------------------------------------------*/


typedef struct window {         /* surface on which to project visible lines*/
        double XMIN, XMAX, YMIN, YMAX,
            ZMIN, ZMAX;
} WINDOW;


/*-------------------------------------------------------------------*/


/*ORIENTATION within a world:

                        Y
                              0



          -X 90                                -90      X



                              180
                              -Y


NOTE:  sin and cos functions use radians as input
*/


/* this function resides in file: visibility.h*/
LINE_HEAD *conduct_visibility_sweep(WORLD*,double,double,double);
```

```
/*************************************************************************
  The following display functions were used in debugging. They have been
left here to aid in future inspection of variables*/

void display_window(W)
    WINDOW *W;
{
 int DUMMY;

        printf("\n\nWindow limits calculated: ");
        printf("\nX: %.2lf-%.2lf\nY: %.2lf-%.2lf\nZ: %.2lf-%.2lf\n\n",
                W->XMIN,W->XMAX,W->YMIN,W->YMAX,W->ZMIN,W->ZMAX);
        fflush(stdout);
        printf("\n\nEnter a number to continue");
}

void lprint_l(L)
        LINE *L;
{
        printf("\n\nline: X1 = %.2lf Y1 = %.2lf Z1 = %.2lf ",L->X1,L->Y1,L->Z1);
        printf("\n      X2 = %.2lf Y2 = %.2lf Z2 = %.2lf \n",L->X2,L->Y2,L->Z2);
        fflush(stdout);
}

void lprint_llist(LIST)
        LINE_HEAD *LIST;
{
        LINE *NEXT_L=LIST->VLINE_LIST;

        printf("\n\n\nVertical lines (%d) are:\n\n",LIST->VERT_LINES);
        while (NEXT_L) {
                lprint_l(NEXT_L);
                NEXT_L=NEXT_L->NEXT;
        }
        printf("\n\n\nnon-vertical lines (%d) are:\n\n",LIST->LINES);
        fflush(stdout);
        NEXT_L=LIST->LINE_LIST;
        while (NEXT_L) {
                lprint_l(NEXT_L);
                NEXT_L=NEXT_L->NEXT;
        }
}

void print_line(L)
        LINE *L;
{
        printf("\nX1:%.4lf Y1:%.4lf Z1:%.4lf X2:%.4lf Y2:%.4lf Z2:%.4lf ",
                L->X1,L->Y1,L->Z1,L->X2,L->Y2,L->Z2);
}   /* end print_line */
void print_line_list(LH)
        LINE_HEAD *LH;
{
        LINE *NEXT_L;

        NEXT_L=LH->LINE_LIST;
        printf("\n\nThere are %d lines: \n\n",LH->LINES);
        while (NEXT_L) {
                print_line(NEXT_L);
                NEXT_L=NEXT_L->NEXT;
        }
}
```

```
/*************************************************************************/

/* Determines absolute values for doubles. */

double myabs(X)
    double X;
{
  if (X < 0.0)
    X=0.0-X;
  return X;
}


/* Find what z coordinate of the vertex, V, from the model */

float find_z(PH,V)
    POLYHEDRON *PH;   /*parent polyhedron*/
    VERTEX *V;
{
  POLYGON *NEXT_PG;
  VERTEX  *NEXT_V;
  float Z_VALUE=66.6;
  int FOUND=0,PG_CNT=0;


  NEXT_PG=PH->POLYGON_LIST;
  while ((NEXT_PG)&&(FOUND==0)) {   /*loop until parent polygon is found*/
    PG_CNT++;
    NEXT_V=NEXT_PG->VERTEX_LIST;
    while ((NEXT_V)&&(FOUND==0)) {  /*loop until we find the vertex*/
      if (NEXT_V==V) {
            Z_VALUE=NEXT_PG->Z_VALUE;
        FOUND=1;
      }
      NEXT_V=NEXT_V->NEXT;
    } /* end while */
    NEXT_PG=NEXT_PG->NEXT;
  } /* end while */
  return (Z_VALUE);    /*return the z height of V*/
} /* end find_z */

/* calculate where the viewing window lies in model coordinates*/
WINDOW *calc_window(X,Y,Z,ORIENT,FOCAL_LEN)

        double X,Y,Z,ORIENT,FOCAL_LEN;
{
        WINDOW *WIN;
        double  HYP;

  WIN=(WINDOW *)malloc(sizeof(WINDOW));

        HYP = FOCAL_LEN/cos(VIEW_ANGLE/2.0);
        WIN->YMIN = Y+cos(90.0*PI/180.0-ORIENT-VIEW_ANGLE/2.0)* HYP;
        WIN->YMAX = Y+sin(ORIENT-VIEW_ANGLE/2.0) * HYP;
        WIN->XMIN = X+sin(90.0*PI/180.0-ORIENT-VIEW_ANGLE/2.0)* HYP;
        WIN->XMAX = X+cos(ORIENT-VIEW_ANGLE/2.0) * HYP;
        WIN->ZMIN = Z-CCD/2.0;
        WIN->ZMAX = Z+CCD/2.0;
        return WIN;
} /* end calc_window */
```

```
/* Deallocate the memory uses in a line list*/

void free_lines(LH)
        LINE_HEAD *LH;
{
        LINE *NEXT_L, *TRASH;  /*TRASH is temporary variable for freeing*/

        NEXT_L=LH->LINE_LIST;
        while (NEXT_L) {
                TRASH=NEXT_L;
                NEXT_L=NEXT_L->NEXT;
                free(TRASH);
        }
        NEXT_L=LH->VLINE_LIST;
        while (NEXT_L) {
                TRASH=NEXT_L;
                NEXT_L=NEXT_L->NEXT;
                free(TRASH);
        }
        free(LH);                       /* free parent structure */
}   /* end free_lines */


LINE_HEAD *create_line_head()
{
        LINE_HEAD *LH;

        if ((LH=(LINE_HEAD *)malloc(sizeof(LINE_HEAD)))==NULL)
                printf("\n\ncannot create line head\n");
        LH->LINES = 0;
        LH->LINE_LIST = NULL;
        LH->TAIL = NULL;
        return LH;
}   /* end create_line_head */
/* Called by get_full_view to pull lines from the 2d+ model*/

LINE *make_line(I,V1,V2,Z1,Z2)
        INSTANCE *I;
        VERTEX *V1, *V2;
        double   Z1,Z2;
{
        LINE *NEW_LINE;
        double LOCAL_X,LOCAL_Y, ROT_X,ROT_Y, RADS;


  NEW_LINE = (LINE *)malloc(sizeof(LINE));
  NEW_LINE->NEXT = NULL;

  /* adjust all local coordinates to pivot point*/
  LOCAL_X = V1->X - I->PIVOT_X;
  LOCAL_Y = V1->Y - I->PIVOT_Y;

  /* rotate about the z axis */
  RADS = I->ROTATION * PI / 180.0 ;   /* convert degs to rads */
  ROT_X = (cos(RADS)*LOCAL_X)+(sin(RADS)*LOCAL_Y);
  ROT_Y = (cos(RADS)*LOCAL_Y)-(sin(RADS)*LOCAL_X);

  /* translate to proper position in world model */

  NEW_LINE->X1 = I->X + ROT_X;
  NEW_LINE->Y1 = I->Y + ROT_Y;
  NEW_LINE->Z1 = I->Z + Z1;
```

```
  /* calc second vertex */
  LOCAL_X = V2->X - I->PIVOT_X;
  LOCAL_Y = V2->Y - I->PIVOT_Y;

  /* rotate about the z axis */
  RADS = I->ROTATION * PI / 180.0 ;   /* convert degs to rads */
  ROT_X = (cos(RADS)*LOCAL_X)+(sin(RADS)*LOCAL_Y);
  ROT_Y = (cos(RADS)*LOCAL_Y)-(sin(RADS)*LOCAL_X);

  /* translate to proper position in world model */
  NEW_LINE->X2 = I->X + ROT_X;
  NEW_LINE->Y2 = I->Y + ROT_Y;
  NEW_LINE->Z2 = I->Z + Z2;

  return NEW_LINE;
}   /* end make_line */


void add_lines(LIST,L)

        LINE_HEAD *LIST;
        LINE      *L;
{
        LINE *NEXT_LINE;

        if (LIST->LINE_LIST==NULL) {   /*add 1st line to empty list*/
          LIST->LINE_LIST=L;
          LIST->TAIL=L;
          LIST->LINES=1;
        }
        else {                         /*add to end of existing list*/
          LIST->TAIL->NEXT=L;
          LIST->LINES++;
          LIST->TAIL=L;
        }
}   /* end add_lines */



void scale_line(L,SX,SY,SZ)

        LINE *L;
        double SX,SY,SZ;
{
        L->X1 = L->X1 * SX ;
        L->X2 = L->X2 * SX ;
        L->Y1 = L->Y1 * SY ;
        L->Y2 = L->Y2 * SY ;
        L->Z1 = L->Z1 * SZ ;
        L->Z2 = L->Z2 * SZ ;
}   /* end scale_line */
```

```
void scale_window(W,SX,SY,SZ)

        WINDOW *W;
        double SX,SY,SZ;
{
        W->XMIN = W->XMIN * SX ;
        W->XMAX = W->XMAX * SX ;
        W->YMIN = W->YMIN * SY ;
        W->YMAX = W->YMAX * SY ;
        W->ZMIN = W->ZMIN * SZ ;
        W->ZMAX = W->ZMAX * SZ ;
}   /* end scale_line */



/* shift from world coordinates to machine coordinates */

void shift_coord_line(L)
    LINE *L;
{
  double TEMP1, TEMP2;

  TEMP1 = L->Z1;
  TEMP2 = L->Z2;
  L->Z1 = L->X1;
  L->Z2 = L->X2;
  L->X1 = L->Y1;
  L->X2 = L->Y2;
  L->Y1 = TEMP1;
  L->Y2 = TEMP2;
} /* end shift_coord_line */


/* shift from world coordinates to machine coordinates */

void shift_coord_window(W)
    WINDOW *W;
{
  double TEMP1, TEMP2;

/* Z=X  Y=Z   X=Y */
  TEMP1 = W->ZMIN;
  TEMP2 = W->ZMAX;
  W->ZMIN = W->XMIN;
  W->ZMAX = W->XMAX;
  W->XMIN = W->YMIN;
  W->XMAX = W->YMAX;
  W->YMIN = TEMP1;
  W->YMAX = TEMP2;
} /* end shift_coord_window */
```

```
/* translates a line to reflect a new origin (X,Y,Z) */

void translate_line(L,X,Y,Z)

        LINE *L;
        double X,Y,Z;
{
        L->X1 += X;
        L->X2 += X;
        L->Y1 += Y;
        L->Y2 += Y;
        L->Z1 += Z;
        L->Z2 += Z;

}  /* END TRANSLATE_LINE */


/* translates a window to reflect a new origin (X,Y,Z) */

void translate_window(W,X,Y,Z)

        WINDOW *W;
        double X,Y,Z;
{
        W->XMIN += X;
        W->XMAX += X;
        W->YMIN += Y;
        W->YMAX += Y;
        W->ZMIN += Z;
        W->ZMAX += Z;

}  /* END TRANSLATE_WINDOW */



/* rotate about the vertical axis */

void rot_z(L,ORIENT)

        LINE *L;
        double ORIENT;
{
    double X1=L->X1,
           X2=L->X2,
           Y1=L->Y1,
           Y2=L->Y2;

        L->X1 = X1*cos(ORIENT)-Y1*sin(ORIENT);
        L->X2 = X2*cos(ORIENT)-Y2*sin(ORIENT);
        L->Y1 = Y1*cos(ORIENT)+X1*sin(ORIENT);
        L->Y2 = Y2*cos(ORIENT)+X2*sin(ORIENT);

}  /* end rot_z */
```

```
/* rotate the window about the vertical axis */

void rot_window(W,ORIENT)

        WINDOW *W;
    double ORIENT;
{
    double XMIN=W->XMIN,
           XMAX=W->XMAX,
           YMIN=W->YMIN,
           YMAX=W->YMAX;

        W->XMIN = XMIN*cos(ORIENT)-YMIN*sin(ORIENT);
        W->XMAX = XMAX*cos(ORIENT)-YMAX*sin(ORIENT);
        W->YMIN = YMIN*cos(ORIENT)+XMIN*sin(ORIENT);
        W->YMAX = YMAX*cos(ORIENT)+XMAX*sin(ORIENT);

}  /* end rot_z */



/* adjust size of line to reflect change in size due to
   distance from the viewing window's plane*/

void perspective_transform(L,ZMIN)
        LINE *L;
    double ZMIN;
{
    double W1=L->Z1/ZMIN ,W2=L->Z2/ZMIN;

  if (W1!=0.0) {
        L->X1=L->X1/W1;
        L->Y1=L->Y1/W1;
        L->Z1=L->Z1/W1;
     }
   else
     printf("\nERROR --- tried to divide by W1=0\n");
   if (W2!=0.0) {
        L->X2=L->X2/W2;
        L->Y2=L->Y2/W2;
        L->Z2=L->Z2/W2;
     }
   else
     printf("\nERROR --- tried to divide by W2=0\n");
}  /* end perspective_transform */
```

```
/* Calculate the clipping codes for line L. */

void get_clipping_codes(L,ZMIN)
    LINE *L;
    double ZMIN;
{
    int i;

    for (i=0;i<=5;++i) {
      L->CLIP1[i]=0;
      L->CLIP2[i]=0;
    }
    if (L->Y1>-L->Z1)
      L->CLIP1[0]=1;
    if (L->Y1<L->Z1)
      L->CLIP1[1]=1;
    if (L->X1>-L->Z1)
      L->CLIP1[2]=1;
    if (L->X1<L->Z1)
      L->CLIP1[3]=1;
    if (L->Z1<-1.0)
      L->CLIP1[4]=1;
    if (L->Z1>ZMIN)
      L->CLIP1[5]=1;
    if (L->Y2>-L->Z2)
      L->CLIP2[0]=1;
    if (L->Y2<L->Z2)
      L->CLIP2[1]=1;
    if (L->X2>-L->Z2)
      L->CLIP2[2]=1;
    if (L->X2<L->Z2)
      L->CLIP2[3]=1;
    if (L->Z2<-1.0)
      L->CLIP2[4]=1;
    if (L->Z2>ZMIN)
      L->CLIP2[5]=1;
  }  /* end get_clipping_codes */


/* Clipt will determine new increments (TE and TL) along line
   being clipped */

void clipt(NUM,DENOM,TE,TL)
    double NUM, DENOM;
    double *TE, *TL;
{
  double t;

  if (DENOM<0.0) {
    t=NUM/DENOM;
    if (t>*TL)
      t=t;
    else
      if (t>*TE)
            *TE=t;
  }
  if (DENOM>0.0) {
    t=NUM/DENOM;
    if (t<*TE)
      t=t;
    else
      if (t<*TL)
```

```
            *TL=t;
  }
}   /* end clipt */

/* Parametric equations of line are used to clip it against the
   canonical view volume*/

void clip_line(L,ZMIN)
     LINE *L;
     double ZMIN;
{
  double dx, dy, dz;
  double TMIN=0.0, TMAX=1.0;


  dx=L->X2-L->X1;
  dy=L->Y2-L->Y1;
  dz=L->Z2-L->Z1;

  clipt((-L->X1-L->Z1),(dx+dz),&TMIN,&TMAX);
  clipt((L->X1-L->Z1),(-dx+dz),&TMIN,&TMAX);
  clipt((L->Y1-L->Z1),(-dy+dz),&TMIN,&TMAX);
  clipt((-L->Y1-L->Z1),(dy+dz),&TMIN,&TMAX);
  clipt((-L->Z1+ZMIN),(dz),&TMIN,&TMAX);
  clipt((-L->Z1-1),(-dz),&TMIN,&TMAX);
  if (TMAX<1) {                    /* endpoint adjusted */
    L->X2 = L->X1 + (TMAX*dx);
    L->Y2 = L->Y1 + (TMAX*dy);
    L->Z2 = L->Z1 + (TMAX*dz);
  }                                /* endpoint adjusted */
  if (TMIN>0) {
    L->X1 = L->X1 + (TMIN*dx);
    L->Y1 = L->Y1 + (TMIN*dy);
    L->Z1 = L->Z1 + (TMIN*dz);
  }
}   /* end clip_line */


/* COMPARES POSITION OF LINE TO VIEW VOLUME:
   returned codes:    0 outside of view volume
                      1 partially inside volume
                      2 entirely in view volume
*/

int clip_line_3d(L)
          LINE *L;
{
          int       IN_VOLUME=1, i, C1=0 ,C2=0;

          for (i=0;i<=5;++i) {
            C1+=L->CLIP1[i];
            C2+=L->CLIP2[i];
            if ((L->CLIP1[i]==1)&&(L->CLIP2[i]==1))
              IN_VOLUME=0;   /* outside view volume */
          }
          if ((IN_VOLUME==1)&&((C1==0)&&(C2==0)))
            IN_VOLUME=2;   /* entirely in view volume */
          return IN_VOLUME;
}   /* end clip_line_3d */
```

```
/* Maps the final line coordinates (from the canonical volume) to
   the desired destination device coordinates.
   MAX_X and MAX_Y are declared at the top of this file and can be
   modified as needed*/

void map_to_screen(L,XMIN,YMIN)
    LINE *L;
    double XMIN,YMIN;
{
 L->X1 = myabs((L->X1-XMIN)/(2*XMIN)*MAX_X);
 L->X2 = myabs((L->X2-XMIN)/(2*XMIN)*MAX_X);
 L->Y1 = myabs((L->Y1-YMIN)/(2*YMIN)*MAX_Y);
 L->Y2 = myabs((L->Y2-YMIN)/(2*YMIN)*MAX_Y);

/*  standard limits on iris are: 1279.0 , 1023.0 */
}  /* end map_to_screen */



/* A raw line goes thru the normalizing transformation and clipping.
   A 1 is returned if line was not totally clipped out of view */

int project_line(X,Y,Z,ORIENT,L,W,W1,FL)

        double X,Y,Z,ORIENT;
        LINE *L;
        WINDOW *W,*W1;
    double FL;
{
        double ZMIN,SCALEX,SCALEY,SCALEZ,VRP_Z;
        int   USED_LINE=1, CLIPT;
        double fl=1.24;
        double X1,Y1,Z1,XTEMP,YTEMP;

        translate_line(L,-W->XMIN,-W->YMIN,-W->ZMIN); /*Make VRP origin*/
        rot_z(L,-ORIENT);
        X1=X-W->XMIN;      /*TRANSLATE and rotate the camera position*/
        Y1=Y-W->YMIN;
        Z1=Z-W->ZMIN;
        XTEMP=X1;
        YTEMP=Y1;
        X1 = XTEMP*cos(-ORIENT)-YTEMP*sin(-ORIENT);
        Y1 = YTEMP*cos(-ORIENT)+XTEMP*sin(-ORIENT);
        translate_line(L,-X1,-Y1,-Z1);
/* change from world to view coords */

/* shear so view volume centered on z-axis is not needed*/

/* now scale view vol to unity using s_per */
/* NOTE: FAR_CLIP is global value */
        VRP_Z = -Y1; /*since still in world coords*/
        SCALEX = 2.0*VRP_Z/((W1->XMAX-W1->XMIN)*(VRP_Z+FARCLIP));
        SCALEY = 2.0*VRP_Z/((W1->YMAX-W1->YMIN)*(VRP_Z+FARCLIP));
        SCALEZ = -1.0/(VRP_Z+FARCLIP);
        shift_coord_line(L);
        scale_line(L,SCALEX,SCALEY,SCALEZ);
        ZMIN=SCALEZ*(VRP_Z+NEARCLIP);
        get_clipping_codes(L,ZMIN);
        CLIPT=clip_line_3d(L);          /*see if any of line is showing*/
        if (CLIPT!=0) {                 /*if so clip off unwanted parts*/
            if (CLIPT==1)
                clip_line (L,ZMIN);
```

```
                perspective_transform(L,ZMIN);   /*project onto window*/
                map_to_screen(L,ZMIN,ZMIN);      /*map to device coords*/
        }
        else
                USED_LINE=0;
        return USED_LINE;           /*let caller know if line accepted or not*/
}   /* end project_line */


/* Remove unwanted lines from final list. Notice that this is only used by
get_view to filter out the set of line returned from conduct_visibility_sweep*/

void remove_line(L,LH)
        LINE *L;
        LINE_HEAD *LH;
{
        LINE *NEXT_L=LH->LINE_LIST, *TRASH;

        if (L==LH->LINE_LIST) {
                LH->LINE_LIST=LH->LINE_LIST->NEXT;
                free(L);
        }
        else {
                while ((NEXT_L->NEXT)&&(NEXT_L->NEXT!=L)) {
                        NEXT_L=NEXT_L->NEXT;
                }
                NEXT_L->NEXT=NEXT_L->NEXT->NEXT;
                free(L);
        }
        LH->LINES--;
}  /* end remove_line */



void remove_vert_line(L,LH)
        LINE *L;
        LINE_HEAD *LH;
{
        LINE *NEXT_L=LH->VLINE_LIST, *TRASH;

        if (L==LH->VLINE_LIST) {
                LH->VLINE_LIST=LH->VLINE_LIST->NEXT;
                free(L);
        }
        else {
                while ((NEXT_L->NEXT)&&(NEXT_L->NEXT!=L)) {
                        NEXT_L=NEXT_L->NEXT;
                }
                NEXT_L->NEXT=NEXT_L->NEXT->NEXT;
                free(L);
        }
        LH->VERT_LINES--;
}  /* end remove_vert_line */
```

```
/* Called from outside the file. This function calls conduct_visibility_sweep
(file visibility.h) to generate a list of lines which may be visible from
the camera position (PRPX,PRPY,PRPZ). The returned lines are then inspected to
determine if they fall within the cameras field of vision (view volume). Those
that don't are removed from the list. Those that are seen are projected into
2d coodinates and mapped to an output device (i.e.- an iris screen). The camera
field of vision is determined by focal length (FL) parameter and the CCD size
declared at the top of this file.
  INPUT:   camera position          PRPX,PRPY,PRPZ
           camera orientatiuon      ORIENT  (0.0 is down y-axis of model)
           target world pointer     W
           camera focal length      FL
*/
LINE_HEAD *get_view(PRPX,PRPY,PRPZ,ORIENT,W,FL)

        double PRPX,PRPY,PRPZ,ORIENT,FL;
        WORLD *W;
{
        LINE        *NEXT_L,*TRASH;
        LINE_HEAD  *LH;          /*list of visible lines*/
        WINDOW     *WIN, *WI;
        double     Z1, Z2, XX, YY, ZZ, XTEMP, YTEMP;
        int            count=0;

        VIEW_ANGLE=2.0*atan(CCD/(2.0*FL));
        ORIENT=ORIENT*PI/180.0;  /*convert to rads*/
        WIN=calc_window(PRPX,PRPY,PRPZ,ORIENT,FL); /*2nd window for reference*/
        WI=calc_window(PRPX,PRPY,PRPZ,ORIENT,FL);
        translate_window(WI,-(WIN->XMIN),-(WIN->YMIN),
                         -(WIN->ZMIN));
        rot_window(WI,-ORIENT);
        XX=PRPX-WIN->XMIN;          /*calculate PRP'*/
        YY=PRPY-WIN->YMIN;
        ZZ=PRPZ-WIN->ZMIN;
        XTEMP=XX;
        YTEMP=YY;
        XX = XTEMP*cos(-ORIENT)-YTEMP*sin(-ORIENT);
        YY = YTEMP*cos(-ORIENT)+XTEMP*sin(-ORIENT);
        translate_window(WI,-XX,-YY,-ZZ);
/*shift from model to graphics coordinates*/
        shift_coord_window(WI);
/*Get the set of all lines which may be visible*/
        LH=conduct_visibility_sweep(W,PRPX,PRPY,PRPZ);
        NEXT_L=LH->VLINE_LIST;
        while (NEXT_L) {
                if (project_line(PRPX,PRPY,PRPZ,ORIENT,NEXT_L,WIN,WI,FL)!=1){
                        TRASH=NEXT_L;
                        NEXT_L=NEXT_L->NEXT;
                        remove_vert_line(TRASH,LH);   /*delete unseen lines*/
                }
                else        {
                        NEXT_L=NEXT_L->NEXT;
                }
        } /* end while */
        NEXT_L=LH->LINE_LIST;
        while (NEXT_L) {
                if (project_line(PRPX,PRPY,PRPZ,ORIENT,NEXT_L,WIN,WI,FL)!=1){
                        TRASH=NEXT_L;
                        NEXT_L=NEXT_L->NEXT;
                        remove_line(TRASH,LH);       /*delete unseen lines*/
                }
                else        {
```

```
                        NEXT_L=NEXT_L->NEXT;
                    }
            } /* end while */
            free(WIN);          /*deallocate memory*/
            free(W1);
            return LH;      /*return list of lines seen (in final device coords*/
}   /* end get_view */


/* This function operates exactly like get_view except that no call is made
   to conduct_visiblity_sweep. Instead the model is stepped through and
   make_line is called to construct each line from the model. The resulting
   output is a list of all model lines (as if everything was transparent).*/

LINE_HEAD *get_full_view(PRPX,PRPY,PRPZ,ORIENT,W,FL)

            double PRPX,PRPY,PRPZ,ORIENT,FL;
            WORLD *W;
{
            POLYHEDRON *NEXT_PH;
            POLYGON    *NEXT_PG;
            VERTEX     *NEXT_V;
        INSTANCE   *NEXT_I;
            LINE        *NEXT_L;
            LINE_HEAD  *LH=create_line_head();
            WINDOW     *WIN, *W1;
            double      Z1, Z2, XX, YY, ZZ, XTEMP, YTEMP;
            int         count=0;

            ORIENT=(ORIENT-0.0)*PI/180.0;   /*convert to rads*/
            VIEW_ANGLE=2.0*atan(CCD/(2.0*FL));
            WIN=calc_window(PRPX,PRPY,PRPZ,ORIENT,FL);
            W1=calc_window(PRPX,PRPY,PRPZ,ORIENT,FL);
            translate_window(W1,-(WIN->XMIN),-(WIN->YMIN),
                                -(WIN->ZMIN));
            rot_window(W1,-ORIENT);
            XX=PRPX-WIN->XMIN;
            YY=PRPY-WIN->YMIN;
            ZZ=PRPZ-WIN->ZMIN;
            XTEMP=XX;
            YTEMP=YY;
            XX = XTEMP*cos(-ORIENT)-YTEMP*sin(-ORIENT);
            YY = YTEMP*cos(-ORIENT)+XTEMP*sin(-ORIENT);
            translate_window(W1,-XX,-YY,-ZZ);
/* change from world to view coords */
            shift_coord_window(W1);
            NEXT_PH=W->POLYHEDRON_LIST;
            while (NEXT_PH) {
              NEXT_I=NEXT_PH->INSTANCE_LIST;
              while (NEXT_I) {
                    NEXT_PG=NEXT_PH->POLYGON_LIST;
                    while(NEXT_PG) {
                            NEXT_V=NEXT_PG->VERTEX_LIST;
                            Z1=NEXT_PG->Z_VALUE;
                            while(NEXT_V) {
                              if (NEXT_V->VERT_EDGE) {
                                Z2=find_z(NEXT_PH,NEXT_V->VERT_EDGE);
                        NEXT_L=make_line(NEXT_I,NEXT_V,NEXT_V->VERT_EDGE,Z1,Z2);
                if (project_line(PRPX,PRPY,PRPZ,ORIENT,NEXT_L.WIN,W1,FL)==1){
                                        add_lines(LH,NEXT_L);
                    }
                            } /* end if */
```

```
                        if (NEXT_V->NEXT) {
                         NEXT_L=make_line(NEXT_I,NEXT_V,NEXT_V->NEXT,Z1,Z1);
          if (project_line(PRPX,PRPY,PRPZ,ORIENT,NEXT_L,WIN,W1,FL)==1){
                                        add_lines(LH,NEXT_L);
                }
                           NEXT_V=NEXT_V->NEXT;
                        } /* end if */
                        else {
              NEXT_L=make_line(NEXT_I,NEXT_V,NEXT_PG->VERTEX_LIST,Z1,Z1);
           if (project_line(PRPX,PRPY,PRPZ,ORIENT,NEXT_L,WIN,W1,FL)==1){
                                        add_lines(LH,NEXT_L);
                }
                           NEXT_V=NULL;
                        } /* end else */
                      } /* end while */
                      NEXT_PG=NEXT_PG->NEXT;
                } /* end while */
                NEXT_I=NEXT_I->NEXT;
          } /* end while */
          NEXT_PH=NEXT_PH->NEXT;
        } /* end while */
        free(WIN);        /*deallocate memory*/
        free(W1);
        return LH;
} /* end get_full_view */
```

```
/*********************************************************************
FILE NAME:    2d+sim.h
AUTHOR:       LT James Stein
PROJECT:      Thesis, wire frame simulator for YAMIBICO
Date:         Mar 1992

Calls to file(s): graphics.h
                  -lgl     (general iris graphics library)

  This program is used to display a world which has been created through
uses of the 2d+ model construction functions in file '2d+.c'. Objects in
the world are drawn to the screen as wire frames as seen from the current
robot configuration in the world.

  The simulator currently gives you control of robot (eye)
movement through use of the mouse. The middle button provides a menu of
options for increasing/decreasing speed, pausing the simulation, starting
the simulation, and quitting. The robots direction is limited to the X/Z
plane and is controlled by the left/right mouse buttons.

  The simulator must be passed a pointer to a WORLD structure when called.
A query is then sent to the user to supply the initial configuaration
of the robot within this world.
*********************************************************************/

typedef struct config
{
  double X,Y,Z;
  double THETA;
} CONFIG;

#define ASPECT       1.25     /*aspect ratio for display window*/
#define FOCAL_LEN    1.24     /*camera's focal length*/
#define VIEW_FIELD   300.0    /*in tenths of degrees*/




/* Get the initial position and heading (configuaration) of the robot from user*/


CONFIG *get_initial_config()

{
  CONFIG *START_CONFIG;
  double DEGS;

  START_CONFIG=(CONFIG *)malloc(sizeof(CONFIG));
  printf("\nEnter initial configuration of robot:\n");
  printf("X: ");
  scanf("\n%lf",&START_CONFIG->X);
  printf("\nY: ");
  scanf("\n%lf",&START_CONFIG->Y);
  printf("\nZ (height of eye): ");
  scanf("\n%lf",&START_CONFIG->Z);
  printf("\nEnter angle of orientation in X/Z plane (in degrees): ");
  scanf("\n%lf",&START_CONFIG->THETA);
  return START_CONFIG;
}   /* end get_initial_config */
```

```
void print_intro()
{
  printf("\n\nIntroduction to the YAMIBICO simulator:\n");
  printf("\n\nThis simulator will display a robot's eye view of a world");
  printf("\nwhich has been constructed in the 2d+ format.");
  printf("\n\nThe world is displayed as a wire frame model and you can");
  printf("\ncontrol the walkthru's speed and motion.\n");
}  /* end print_intro */


void print_instructions()
{
  printf("\n\nINSTRUCTIONS:\n");
  printf("\nYou will need to enter the starting position of the robot ");
  printf("\nin your world. The robots heading can be controlled by ");
  printf("\nhitting the left/right mouse buttons. The middle mouse ");
  printf("\nbutton will present you with a menu of other options for: ");
  printf("\n   -controlling speed ");
  printf("\n   -pausing simulation ");
  printf("\n   -starting simulation ");
  printf("\n   -quitting simulation");
  printf("\n\n For now enter the starting position of your robot and ");
  printf("\nthe theta angle in the X/Z plane: ");

}  /* end print_instructions */



/* initialize the window parameters */
void initialize()
{
  winopen("WORLD VIEW");
  wintitle("5th floor");
  doublebuffer();
  RGBmode();
  qdevice(REDRAW);
  qdevice(WINQUIT);
  qdevice(WINSHUT);
  qdevice(LEFTMOUSE);
  qdevice(MIDDLEMOUSE);
  qdevice(RIGHTMOUSE);
}   /* end of initialize */



/* define menus which can be presented to the user */
int define_menus()
{
  int MAIN_MENU,POLY_MENU,SELECT_MENU;

MAIN_MENU=defpup("OPTIONS:%t|START/RESTART|PAUSE|SLOWER|FASTER|STOP|QUIT%x99");
  return (MAIN_MENU);
}  /* end define_menus */
```

```
/*****************************************************************
 *          FUNCTION:          set_color()
 *              used to set the RGB color of display
 *****************************************************************/

void set_color(index)
        int index;              /* color index from building array          */
{

        switch(index) {
                case 0:     RGBcolor(0, 0, 0);
                            break;      /* black                */
                case 1: RGBcolor(255, 255, 255);
                             break;  /*  white */
                case 3:     RGBcolor(0, 150, 0);
                            break;      /* green */
                case 4: RGBcolor(0, 0, 245);
                        break;  /* blue */
                default:
                  printf("error in color coding");
        }
}

/* copy a new configuration into an old one */

translate_config(WORLD_CONFIG,NEW_CONFIG)
    CONFIG *WORLD_CONFIG, *NEW_CONFIG;
{
  NEW_CONFIG->X=WORLD_CONFIG->X;
  NEW_CONFIG->Y=WORLD_CONFIG->Y;
  NEW_CONFIG->Z=WORLD_CONFIG->Z;
  NEW_CONFIG->THETA=WORLD_CONFIG->THETA;
}


/* set up viewing situation in 3d environment*/

void proj_view_matrix(WORLD_CONFIG)
    CONFIG *WORLD_CONFIG;
{
  douhle REFX,REFZ;
  CONFIG *NEW_CONFIG;

  perspective(VIEW_FIELD,ASPECT,NEARCLIP,FARCLIP);
  NEW_CONFIG=(CONFIG *)malloc(sizeof(CONFIG));
  translate_config(WORLD_CONFIG,NEW_CONFIG);
  REFX=(NEW_CONFIG->X+cos(NEW_CONFIG->THETA));
  REFZ=NEW_CONFIG->Z+sin(NEW_CONFIG->THETA);
  lookat(NEW_CONFIG->X,NEW_CONFIG->Y,NEW_CONFIG->Z,
            REFX,NEW_CONFIG->Y,REFZ,0);      /*tell system the position of eye*/
  free(NEW_CONFIG);
}   /* end proj_view_matrix */



/* project movement of robot along current theta in proportion to
   current velocity */
```

```
calc_config(OLD_CONFIG,VELOCITY,NEW_CONFIG)
    CONFIG *OLD_CONFIG, *NEW_CONFIG;
    double  VELOCITY;
{

  NEW_CONFIG->THETA=OLD_CONFIG->THETA;
  NEW_CONFIG->Z=OLD_CONFIG->Z;
  NEW_CONFIG->X=OLD_CONFIG->X-VELOCITY*sin(OLD_CONFIG->THETA*PI/180.0);
  NEW_CONFIG->Y=OLD_CONFIG->Y+VELOCITY*cos(OLD_CONFIG->THETA*PI/180.0);
}   /* end calc_config */



/* print current configuration onto the screen */

void display_config(ORIENTATION)
    CONFIG *ORIENTATION;
{
  char *MSG;
  CONFIG *DISPLAY;
  double MOCK_V=5.0;

  DISPLAY=(CONFIG *)malloc(sizeof(CONFIG));
  calc_config(ORIENTATION,MOCK_V,DISPLAY);
  MSG=(char *)calloc(80,sizeof(char));
  sprintf(MSG,"X: %.2lf  Y: %.2lf  Z: %.2lf",
          ORIENTATION->X,ORIENTATION->Y,ORIENTATION->Z);
  cmov2(500.0,100.0);  /*line for coordinates*/
  charstr(MSG);              /*line for orientation*/
  cmov2(500.0,50.0);
  sprintf(MSG,"THETA(degs): %.2lf",ORIENTATION->THETA);
  charstr(MSG);
  free(DISPLAY);
}   /* end display_config */



/* Draw the set of lines extracted from the model to the screen*/

void draw_screen(LIST)
    LINE_HEAD *LIST;
{
    LINE *NEXT_L=LIST->LINE_LIST;

  linewidth(1);
  RGBcolor(0,0,0);
  ortho2(0.0,XMAXSCREEN,0.0,YMAXSCREEN);
  while (NEXT_L) {                    /*draw non-vertical lines*/
    move2(NEXT_L->X1,NEXT_L->Y1);
    draw2(NEXT_L->X2,NEXT_L->Y2);
    NEXT_L=NEXT_L->NEXT;
  }
  NEXT_L=LIST->VLINE_LIST;
  while (NEXT_L) {                    /*draw vertical lines*/
    move2(NEXT_L->X1,NEXT_L->Y1);
    draw2(NEXT_L->X2,NEXT_L->Y2);
    NEXT_L=NEXT_L->NEXT;
  }
}  /* end draw_screen */
```

```
/* Process the option selected from pull down menu
 NOTE: some space is reserved for future functionallity*/

void processmenuhit(CHOICE,RUN_PROGRAM,VELOCITY)
     int CHOICE;
     int *RUN_PROGRAM;
     double *VELOCITY;
{
  switch(CHOICE)  {
  case -1:       /* no selection */
    break;
  case 1:       /* start simulation */
    *RUN_PROGRAM=1;
    break;
  case 2:      /* stop simulation */
    *RUN_PROGRAM=0;
    break;
  case 3:
    *VELOCITY-=10;
     break:    /*slower*/
  case 4:
    *VELOCITY+=20;
     break:    /*faster*/
  case 5:
     *VELOCITY=0;
     break;    /*stop robot*/

/*future use*/

  case 6:      /* pick closest polygon */
    break;
  case 7:      /* pick next polygon */
    break;
  case 8:      /* delete polygon */
    break;
  case 9:      /* modify polygon */
    break;
  case 99:     /* terminate program */
    break;
  default:
    break;
  }   /* end switch on CHOICE */
}   /* end processmenuhit */
```

```
/************************main procedure ********************
  Calls either the get_view or get_full_view function from file
  graphics.c. The 1st will return a list of visible lines from
  THE_WORLD while the latter returns a set of all lines.
*************************************************************/

void simulate(THE_WORLD)
        WORLD *THE_WORLD;
{
 CONFIG *OLD_CONFIG, *NEW_CONFIG;
 int VALUE;
 int MENU_CHOICE=1,MAIN_MENU,RUN_PROGRAM=0,COLLISION=0;
 int *RP;
 double VELOCITY=0;
 double *V;
 LINE_HEAD *LLIST;

 print_intro();
 print_instructions();
 OLD_CONFIG=get_initial_config();  /*get start position and heading*/
 LLIST = get_full_view(OLD_CONFIG->X,OLD_CONFIG->Y,OLD_CONFIG->Z,
                OLD_CONFIG->THETA,THE_WORLD,FOCAL_LEN);/*get visible lines*/
 initialize();
 RP= &RUN_PROGRAM;
 V= &VELOCITY;      /*assign address of velocity to pointer V*/
 MAIN_MENU=define_menus();
 NEW_CONFIG=(CONFIG *)malloc(sizeof(CONFIG));  /*allcoate memory*/
 proj_view_matrix(OLD_CONFIG);
 zbuffer(TRUE);
 RGBcolor(255,255,255);  /*set to white*/
 clear();
 swapbuffers();      /*clear display screen*/
 clear();
 draw_screen(LLIST);  /*draw the view*/
 RGBcolor(0,0,0);
 display_config(OLD_CONFIG);  /*display the starting configuration*/
 zbuffer(FALSE);
 swapbuffers();      /*double buffering to smooth out simulation*/

 while (MENU_CHOICE!=99) {
  if (qtest()) {                /* action is queued */
   switch (qread(&VALUE)) {
    case MIDDLEMOUSE:               /*bring up menu of options*/
        MENU_CHOICE=dopup(MAIN_MENU);
        processmenuhit(MENU_CHOICE,RP,V);  /*go do what user selected*/
        break;
       case LEFTMOUSE:              /*turn left*/
        OLD_CONFIG->THETA=OLD_CONFIG->THETA+5.0;
      if (OLD_CONFIG->THETA>360.0)
                OLD_CONFIG->THETA-=360.0;
        break;
```

```
            case RIGHTMOUSE:            /*turn right*/
              OLD_CONFIG->THETA-=5.0;
              if (OLD_CONFIG->THETA<0.0)
                    OLD_CONFIG->THETA+=360.0;
              break;
            case REDRAW:
              reshapeviewport();
              break;
            case WINQUIT:
              gexit();
              break;
            default:
              break;
            }
        }
        if (RUN_PROGRAM==1) {
          calc_config(OLD_CONFIG,VELOCITY,NEW_CONFIG);/*move IAW velocity*/
          proj_view_matrix(NEW_CONFIG);
          free_lines(LLIST);                /*deallocate memory used last time*/
                                            /*then get the next view*/
          LLIST = get_full_view(NEW_CONFIG->X,NEW_CONFIG->Y,NEW_CONFIG->Z,
                        NEW_CONFIG->THETA,THE_WORLD,FOCAL_LEN);
          zbuffer(TRUE);
          RGBcolor(255,255,255);            /*draw white on black*/
          clear();
          draw_screen(LLIST);               /*draw the view*/
          zbuffer(FALSE);
          swapbuffers();
          RGBcolor(0,0,0);
          display_config(NEW_CONFIG);       /*display the current configuration*/
          RGBcolor(255,255,255);
          OLD_CONFIG->X=NEW_CONFIG->X;      /*update to next configuration*/
          OLD_CONFIG->Y=NEW_CONFIG->Y;
          OLD_CONFIG->Z=NEW_CONFIG->Z;
          OLD_CONFIG->THETA=NEW_CONFIG->THETA;
        }   /* end run program==1 */
      }   /* end while */
      free(OLD_CONFIG);                     /*deallocate last memory used*/
      free(NEW_CONFIG);
      free_lines(LLIST);            /*notice the world is left intact*/
    }  /* end main() */
```

```
/* FILE:          5th.h
   AUTHOR:        LT James Stein
   THESIS ADVISOR: Dr. Kanayama
   CALLS TO FILES: 2d+d.h
   COMMENTS: This is the construction file for the 2d+ model of the
5th floor Spanagel Hall (1st half only - up to glass double doors). All
coordinates are in inches while all angles are in degrees.
          The main function "make_world" is called to
build the model using function calls to file 2d+.h. Type definitions for
WORLD, POLYHEDRON, POLYGON, and VERTEX can be found at the top of this file
also.
   Notice that the floor of H1 is one huge, concave polygon which
makes up the floor to the hallway as well as all of the office floors. To this
floor numerous ceilings are added for offices, door jams, and main corridors.
Doors, lights, and molding strips are then added to the model as separate
polyhedra.
*/

WORLD *make_world()
{
  WORLD *W;
  POLYHEDRON *H1, *H2, *H3, *H4, *H5, *H6, *H7, *H8, *H9, *H10, *H11, *H12,
    *H13, *H14, *H15, *H16, *H17, *H18, *H19, *H20, *H21, *H22, *H23, *H24,
    *H25, *H26, *H27, *H28, *H29, *H30, *H31, *H32, *H33, *H34, *H35, *H36,
    *H37, *H38, *H39, *H40, *H41, *H42, *H43;
  POLYGON *H1P1, *H1P2, *H1P3,
    *H1P4, *H1P5 ,*H1P6, *H1P7, *H1P8, *H1P9, *H1P10, *H1P11, *H1P12,
    *H1P13, *H1P14, *H1P15, *H1P16, *H1P17, *H1P18, *H1P19, *H1P20, *H1P21,
    *H1P22, *H1P23, *H1P24, *H1P25, *H1P26, *H1P27, *H1P28, *H1P29, *H1P30,
    *H1P31, *H1P32, *H1P33, *H1P34, *H1P35, *H1P36, *H1P37, *H1P38, *H1P39,
    *H1P40, *H1P41, *H1P42, *H1P43, *H1P44, *H1P45, *H1P46, *H1P47, *H1P48,
    *H1P49, *H1P50, *H1P51, *H1P52, *H1P53, *H1P54, *H1P55, *H1P56, *H1P57,
    *H1P58, *H1P59, *H1P60, *H1P61, *H1P62, *H1P63, *H1P64, *H1P65,

    *H2P1, *H2P2, *H3P1, *H3P2, *H4P1, *H4P2, *H5P1, *H5P2,
    *H6P1, *H7P1, *H7P2, *H8P1, *H8P2, *H9P1, *H9P2, *H10P1, *H10P2,
    *H11P1, *H11P2, *H12P1, *H12P2, *H13P1, *H13P2, *H14P1, *H14P2, *H15P1,
    *H15P2, *H16P1, *H16P2, *H17P1, *H17P2, *H18P1, *H18P2, *H19P1, *H19P2,
    *H20P1, *H20P2,
    *H21P1, *H21P2, *H22P1, *H22P2, *H23P1, *H23P2, *H24P1, *H24P2, *H25P1,
    *H25P2, *H26P1, *H26P2, *H27P1, *H27P2, *H28P1, *H28P2, *H29P1, *H29P2,
    *H30P1, *H30P2,
    *H31P1, *H31P2, *H32P1, *H32P2, *H33P1, *H33P2, *H34P1, *H34P2, *H35P1,
    *H35P2, *H36P1, *H36P2, *H37P1, *H37P2, *H38P1, *H38P2, *H39P1, *H39P2,
    *H40P1, *H40P2, *H41P1, *H41P2, *H42P1, *H42P2, *H43P1, *H43P2,
last_p;

  VERTEX *H1P1V1, *H1P1V2, *H1P1V3, *H1P1V4, *H1P1V5, *H1P1V6, *H1P1V7,
       *H1P1V8,*H1P1V9,*H1P1V10,*H1P1V11, *H1P1V12, *H1P1V13, *H1P1V14,
       *H1P1V15,*H1P1V16,*H1P1V17,*H1P1V18, *H1P1V19, *H1P1V20, *H1P1V21,
       *H1P1V22,*H1P1V23,*H1P1V24,*H1P1V25, *H1P1V26, *H1P1V27, *H1P1V28,
       *H1P1V29,*H1P1V30,*H1P1V31,*H1P1V32, *H1P1V33, *H1P1V34, *H1P1V35,
       *H1P1V36,*H1P1V37,*H1P1V38,*H1P1V39, *H1P1V40, *H1P1V41, *H1P1V42,
       *H1P1V43,*H1P1V44,*H1P1V45,*H1P1V46, *H1P1V47, *H1P1V48, *H1P1V49,
       *H1P1V50,*H1P1V51,*H1P1V52,*H1P1V53, *H1P1V54, *H1P1V55, *H1P1V56,
       *H1P1V57,*H1P1V58,*H1P1V59,*H1P1V60, *H1P1V61, *H1P1V62, *H1P1V63,
       *H1P1V64,*H1P1V65,*H1P1V66,*H1P1V67, *H1P1V68, *H1P1V69, *H1P1V70,

       *H1P1V2a, *H1P1V2b, *H1P1V2c, *H1P1V2d, *H1P1V2e, *H1P1V2f,
       *H1P1V4a, *H1P1V4b, *H1P1V4c, *H1P1V4d, *H1P1V4e, *H1P1V4f,
       *H1P1V6a, *H1P1V6b, *H1P1V6c, *H1P1V6d, *H1P1V6e, *H1P1V6f,
       *H1P1V8a, *H1P1V8b, *H1P1V8c, *H1P1V8d, *H1P1V8e, *H1P1V8f,
```

*H1P1V10a, *H1P1V10b, *H1P1V10c, *H1P1V10d, *H1P1V10e, *H1P1V10f,
*H1P1V12a, *H1P1V12b, *H1P1V12c, *H1P1V12d, *H1P1V12e, *H1P1V12f,
*H1P1V14a, *H1P1V14b, *H1P1V14c, *H1P1V14d, *H1P1V14e, *H1P1V14f,
*H1P1V16a, *H1P1V16b, *H1P1V16c, *H1P1V16d, *H1P1V16e, *H1P1V16f,
*H1P1V18a, *H1P1V18b, *H1P1V18c, *H1P1V18d, *H1P1V18e, *H1P1V18f,
*H1P1V20a, *H1P1V20b, *H1P1V20c, *H1P1V20d, *H1P1V20e, *H1P1V20f,
*H1P1V22a, *H1P1V22b,
*H1P1V24a, *H1P1V24b, *H1P1V24c, *H1P1V24d, *H1P1V24e, *H1P1V24f,
*H1P1V26a, *H1P1V26b, *H1P1V26c, *H1P1V26d, *H1P1V26e, *H1P1V26f,
*H1P1V28a, *H1P1V28b, *H1P1V28c, *H1P1V28d, *H1P1V28e, *H1P1V28f,
*H1P1V30a, *H1P1V30b, *H1P1V30c, *H1P1V30d, *H1P1V30e, *H1P1V30f,
*H1P1V32a, *H1P1V32b, *H1P1V32c, *H1P1V32d, *H1P1V32e, *H1P1V32f,
*H1P1V34a, *H1P1V34b, *H1P1V34c, *H1P1V34d, *H1P1V34e, *H1P1V34f,
*H1P1V36a, *H1P1V36b, *H1P1V36c, *H1P1V36d, *H1P1V36e, *H1P1V36f,
*H1P1V38a, *H1P1V38b, *H1P1V38c, *H1P1V38d, *H1P1V38e, *H1P1V38f,
*H1P1V40a, *H1P1V40b, *H1P1V40c, *H1P1V40d, *H1P1V40e, *H1P1V40f,
*H1P1V42a, *H1P1V42b, *H1P1V42c, *H1P1V42d, *H1P1V42e, *H1P1V42f,
*H1P1V44a, *H1P1V44b, *H1P1V44c, *H1P1V44d, *H1P1V44e, *H1P1V44f,
*H1P1V46a, *H1P1V46b, *H1P1V46c, *H1P1V46d, *H1P1V46e, *H1P1V46f,
*H1P1V48a, *H1P1V48b, *H1P1V48c, *H1P1V48d, *H1P1V48e, *H1P1V48f,
*H1P1V50a, *H1P1V50b, *H1P1V50c, *H1P1V50d, *H1P1V50e, *H1P1V50f,
*H1P1V52a, *H1P1V52b, *H1P1V52c, *H1P1V52d, *H1P1V52e, *H1P1V52f,
*H1P1V55a, *H1P1V55b, *H1P1V55c, *H1P1V55d, *H1P1V55e, *H1P1V55f,
*H1P1V58a, *H1P1V58b, *H1P1V58c, *H1P1V58d, *H1P1V58e, *H1P1V58f,
*H1P1V60a, *H1P1V60b, *H1P1V60c, *H1P1V60d, *H1P1V60e, *H1P1V60f,
*H1P1V63a, *H1P1V63b, *H1P1V63c, *H1P1V63d, *H1P1V63e, *H1P1V63f,
*H1P1V63g,
*H1P1V65a, *H1P1V65b, *H1P1V65c, *H1P1V65d, *H1P1V65e, *H1P1V65f,
*H1P1V65g,
*H1P1V68a, *H1P1V68b, *H1P1V68c, *H1P1V68d, *H1P1V68e, *H1P1V68f,

*H1P2V1, *H1P2V2, *H1P2V3, *H1P2V4.
*H1P3V1, *H1P3V2, *H1P3V3, *H1P3V4.
*H1P4V1, *H1P4V2, *H1P4V3, *H1P4V4.
*H1P5V1, *H1P5V2, *H1P5V3, *H1P5V4.
*H1P6V1, *H1P6V2, *H1P6V3, *H1P6V4.
*H1P7V1, *H1P7V2, *H1P7V3, *H1P7V4.
*H1P8V1, *H1P8V2, *H1P8V3, *H1P8V4.
*H1P9V1, *H1P9V2, *H1P9V3, *H1P9V4.
*H1P10V1, *H1P10V2, *H1P10V3, *H1P10V4.
*H1P11V1, *H1P11V2, *H1P11V3, *H1P11V4.
*H1P12V1, *H1P12V2, *H1P12V3, *H1P12V4.
*H1P13V1, *H1P13V2, *H1P13V3, *H1P13V4.
*H1P14V1, *H1P14V2, *H1P14V3, *H1P14V4,
*H1P15V1, *H1P15V2, *H1P15V3, *H1P15V4.
*H1P16V1, *H1P16V2, *H1P16V3, *H1P16V4.
*H1P17V1, *H1P17V2, *H1P17V3, *H1P17V4.
*H1P18V1, *H1P18V2, *H1P18V3, *H1P18V4.
*H1P19V1, *H1P19V2, *H1P19V3, *H1P19V4.
*H1P20V1, *H1P20V2, *H1P20V3, *H1P20V4.
*H1P21V1, *H1P21V2, *H1P21V3, *H1P21V4.
*H1P22V1, *H1P22V2, *H1P22V3, *H1P22V4.
*H1P23V1, *H1P23V2, *H1P23V3, *H1P23V4.
*H1P24V1, *H1P24V2, *H1P24V3, *H1P24V4.
*H1P25V1, *H1P25V2, *H1P25V3, *H1P25V4.
*H1P26V1, *H1P26V2, *H1P26V3, *H1P26V4.
*H1P27V1, *H1P27V2, *H1P27V3, *H1P27V4.
*H1P28V1, *H1P28V2, *H1P28V3, *H1P28V4.
*H1P29V1, *H1P29V2, *H1P29V3, *H1P29V4.
*H1P30V1, *H1P30V2, *H1P30V3, *H1P30V4.
*H1P31V1, *H1P31V2, *H1P31V3, *H1P31V4, *H1P31V5.
*H1P32V1, *H1P32V2, *H1P32V3, *H1P32V4, *H1P32V5.

*H1P33V1, *H1P33V2, *H1P33V3, *H1P33V4,
*H1P34V1, *H1P34V2, *H1P34V3, *H1P34V4,
*H1P35V1, *H1P35V2, *H1P35V3, *H1P35V4,
*H1P36V1, *H1P36V2, *H1P36V3, *H1P36V4,
*H1P37V1, *H1P37V2, *H1P37V3, *H1P37V4,
*H1P38V1, *H1P38V2, *H1P38V3, *H1P38V4,
*H1P39V1, *H1P39V2, *H1P39V3, *H1P39V4,
*H1P40V1, *H1P40V2, *H1P40V3, *H1P40V4,
*H1P41V1, *H1P41V2, *H1P41V3, *H1P41V4,
*H1P42V1, *H1P42V2, *H1P42V3, *H1P42V4,
*H1P43V1, *H1P43V2, *H1P43V3, *H1P43V4,
*H1P44V1, *H1P44V2, *H1P44V3, *H1P44V4,
*H1P45V1, *H1P45V2, *H1P45V3, *H1P45V4,
*H1P46V1, *H1P46V2, *H1P46V3, *H1P46V4,
*H1P47V1, *H1P47V2, *H1P47V3, *H1P47V4,
*H1P48V1, *H1P48V2, *H1P48V3, *H1P48V4,
*H1P49V1, *H1P49V2, *H1P49V3, *H1P49V4,
*H1P50V1, *H1P50V2, *H1P50V3, *H1P50V4,
*H1P51V1, *H1P51V2, *H1P51V3, *H1P51V4,
*H1P52V1, *H1P52V2, *H1P52V3, *H1P52V4,
*H1P53V1, *H1P53V2, *H1P53V3, *H1P53V4,
*H1P54V1, *H1P54V2, *H1P54V3, *H1P54V4,
*H1P55V1, *H1P55V2, *H1P55V3, *H1P55V4,
*H1P56V1, *H1P56V2, *H1P56V3, *H1P56V4,
*H1P57V1, *H1P57V2, *H1P57V3, *H1P57V4,
*H1P58V1, *H1P58V2, *H1P58V3, *H1P58V4,
*H1P59V1, *H1P59V2, *H1P59V3, *H1P59V4,
*H1P60V1, *H1P60V2, *H1P60V3, *H1P60V4,
*H1P61V1, *H1P61V2, *H1P61V3, *H1P61V4,
*H1P62V1, *H1P62V2, *H1P62V3, *H1P62V4,
*H1P63V1, *H1P63V2, *H1P63V3, *H1P63V4,
*H1P64V1, *H1P64V2, *H1P64V3, *H1P64V4,
*H1P65V1, *H1P65V2, *H1P65V3, *H1P65V4,

*H2P1V1, *H2P1V2, *H2P1V3, *H2P1V4, *H2P2V1, *H2P2V2, *H2P2V3, *H2P2V4,
*H3P1V1, *H3P1V2, *H3P1V3, *H3P1V4, *H3P2V1, *H3P2V2, *H3P2V3, *H3P2V4,
*H4P1V1, *H4P1V2, *H4P1V3, *H4P1V4, *H4P2V1, *H4P2V2, *H4P2V3, *H4P2V4,
*H5P1V1, *H5P1V2, *H5P1V3, *H5P1V4, *H5P2V1, *H5P2V2, *H5P2V3, *H5P2V4,
*H6P1V1, *H6P1V2, *H6P1V3, *H6P1V4,
*H7P1V1, *H7P1V2, *H7P1V3, *H7P1V4, *H7P2V1, *H7P2V2, *H7P2V3, *H7P2V4,
*H8P1V1, *H8P1V2, *H8P1V3, *H8P1V4, *H8P2V1, *H8P2V2, *H8P2V3, *H8P2V4,
*H9P1V1, *H9P1V2, *H9P1V3, *H9P1V4, *H9P2V1, *H9P2V2, *H9P2V3, *H9P2V4,
*H10P1V1, *H10P1V2, *H10P1V3, *H10P1V4, *H10P2V1, *H10P2V2, *H10P2V3, *H10P2V4,
*H11P1V1, *H11P1V2, *H11P1V3, *H11P1V4, *H11P2V1, *H11P2V2, *H11P2V3, *H11P2V4,
*H12P1V1, *H12P1V2, *H12P1V3, *H12P1V4, *H12P2V1, *H12P2V2, *H12P2V3, *H12P2V4,
*H13P1V1, *H13P1V2, *H13P1V3, *H13P1V4, *H13P2V1, *H13P2V2, *H13P2V3, *H13P2V4,
*H14P1V1, *H14P1V2, *H14P1V3, *H14P1V4, *H14P2V1, *H14P2V2, *H14P2V3, *H14P2V4,
*H15P1V1, *H15P1V2, *H15P1V3, *H15P1V4, *H15P2V1, *H15P2V2, *H15P2V3, *H15P2V4,
*H16P1V1, *H16P1V2, *H16P1V3, *H16P1V4, *H16P2V1, *H16P2V2, *H16P2V3, *H16P2V4,
*H17P1V1, *H17P1V2, *H17P1V3, *H17P1V4, *H17P2V1, *H17P2V2, *H17P2V3, *H17P2V4,
*H18P1V1, *H18P1V2, *H18P1V3, *H18P1V4, *H18P2V1, *H18P2V2, *H18P2V3, *H18P2V4,
*H19P1V1, *H19P1V2, *H19P1V3, *H19P1V4, *H19P2V1, *H19P2V2, *H19P2V3, *H19P2V4,
*H20P1V1, *H20P1V2, *H20P1V3, *H20P1V4, *H20P2V1, *H20P2V2, *H20P2V3, *H20P2V4,
*H21P1V1, *H21P1V2, *H21P1V3, *H21P1V4, *H21P2V1, *H21P2V2, *H21P2V3, *H21P2V4,
*H22P1V1, *H22P1V2, *H22P1V3, *H22P1V4, *H22P2V1, *H22P2V2, *H22P2V3, *H22P2V4,
*H23P1V1, *H23P1V2, *H23P1V3, *H23P1V4, *H23P2V1, *H23P2V2, *H23P2V3, *H23P2V4,
*H24P1V1, *H24P1V2, *H24P1V3, *H24P1V4, *H24P2V1, *H24P2V2, *H24P2V3, *H24P2V4,
*H25P1V1, *H25P1V2, *H25P1V3, *H25P1V4, *H25P2V1, *H25P2V2, *H25P2V3, *H25P2V4,
*H26P1V1, *H26P1V2, *H26P1V3, *H26P1V4, *H26P2V1, *H26P2V2, *H26P2V3, *H26P2V4,
*H27P1V1, *H27P1V2, *H27P1V3, *H27P1V4, *H27P2V1, *H27P2V2, *H27P2V3, *H27P2V4,
*H28P1V1, *H28P1V2, *H28P1V3, *H28P1V4, *H28P2V1, *H28P2V2, *H28P2V3, *H28P2V4,
*H29P1V1, *H29P1V2, *H29P1V3, *H29P1V4, *H29P2V1, *H29P2V2, *H29P2V3, *H29P2V4,

```
*H30P1V1, *H30P1V2, *H30P1V3, *H30P1V4, *H30P2V1, *H30P2V2, *H30P2V3, *H30P2V4,
*H31P1V1, *H31P1V2, *H31P1V3, *H31P1V4, *H31P2V1, *H31P2V2, *H31P2V3, *H31P2V4,
*H32P1V1, *H32P1V2, *H32P1V3, *H32P1V4, *H32P2V1, *H32P2V2, *H32P2V3, *H32P2V4,
*H33P1V1, *H33P1V2, *H33P1V3, *H33P1V4, *H33P2V1, *H33P2V2, *H33P2V3, *H33P2V4,
*H34P1V1, *H34P1V2, *H34P1V3, *H34P1V4, *H34P2V1, *H34P2V2, *H34P2V3, *H34P2V4,
*H35P1V1, *H35P1V2, *H35P1V3, *H35P1V4, *H35P2V1, *H35P2V2, *H35P2V3, *H35P2V4,
*H36P1V1, *H36P1V2, *H36P1V3, *H36P1V4, *H36P2V1, *H36P2V2, *H36P2V3, *H36P2V4,
*H37P1V1, *H37P1V2, *H37P1V3, *H37P1V4, *H37P2V1, *H37P2V2, *H37P2V3, *H37P2V4,
*H38P1V1, *H38P1V2, *H38P1V3, *H38P1V4, *H38P2V1, *H38P2V2, *H38P2V3, *H38P2V4,
*H39P1V1, *H39P1V2, *H39P1V3, *H39P1V4, *H39P2V1, *H39P2V2, *H39P2V3, *H39P2V4,
*H40P1V1, *H40P1V2, *H40P1V3, *H40P1V4, *H40P2V1, *H40P2V2, *H40P2V3, *H40P2V4,
*H41P1V1, *H41P1V2, *H41P1V3, *H41P1V4, *H41P2V1, *H41P2V2, *H41P2V3, *H41P2V4,
*H42P1V1, *H42P1V2, *H42P1V3, *H42P1V4, *H42P2V1, *H42P2V2, *H42P2V3, *H42P2V4,
*H43P1V1, *H43P1V2, *H43P1V3, *H43P1V4, *H43P2V1, *H43P2V2, *H43P2V3, *H43P2V4,
last_v;

  W=add_world("5th_floor",9);
  H1=add_ph("front_hall",10,W,1,0);
  H1P1=add_pg(H1,0.0,1,0);
  H1P1V1 = add_vertex(H1P1,0.0,0.0);
  H1P1V2 = add_vertex(H1P1,0.0,239.5); /*rm 506*/
  H1P1V2a = add_vertex(H1P1,-5.3,239.5);
  H1P1V2b = add_vertex(H1P1,-5.3,203.3);
  H1P1V2c = add_vertex(H1P1,-244.1,203.3);
  H1P1V2d = add_vertex(H1P1,-244.1,309.4);
  H1P1V2e = add_vertex(H1P1,-5.3,309.4);
  H1P1V2f = add_vertex(H1P1,-5.3,275.2);

  H1P1V3 = add_vertex(H1P1,0.0,275.2);
  H1P1V4 = add_vertex(H1P1,0.0,713.7); /*rm 510*/
  H1P1V4a = add_vertex(H1P1,-5.3,713.7);
  H1P1V4b = add_vertex(H1P1,-5.3,677.5);
  H1P1V4c = add_vertex(H1P1,-244.1,677.5);
  H1P1V4d = add_vertex(H1P1,-244.1,783.6);
  H1P1V4e = add_vertex(H1P1,-5.3,783.6);
  H1P1V4f = add_vertex(H1P1,-5.3,749.4);

  H1P1V5 = add_vertex(H1P1,0.0,749.4);
  H1P1V6 = add_vertex(H1P1,0.0,825.9); /* rm 512*/
  H1P1V6a = add_vertex(H1P1,-5.3,825.9);
  H1P1V6b = add_vertex(H1P1,-5.3,789.7);
  H1P1V6c = add_vertex(H1P1,-244.1,789.7);
  H1P1V6d = add_vertex(H1P1,-244.1,895.8);
  H1P1V6e = add_vertex(H1P1,-5.3,895.8);
  H1P1V6f = add_vertex(H1P1,-5.3,861.6);

  H1P1V7 = add_vertex(H1P1,0.0,861.6);
  H1P1V8 = add_vertex(H1P1,0.0,937.5); /* rm 514*/
  H1P1V8a = add_vertex(H1P1,-5.3,937.5);
  H1P1V8b = add_vertex(H1P1,-5.3,901.3);
  H1P1V8c = add_vertex(H1P1,-244.1,901.3);
  H1P1V8d = add_vertex(H1P1,-244.1,1007.4);
  H1P1V8e = add_vertex(H1P1,-5.3,1007.4);
  H1P1V8f = add_vertex(H1P1,-5.3,973.2);

  H1P1V9 = add_vertex(H1P1,0.0,973.2);
  H1P1V10 = add_vertex(H1P1,0.0,1049.7); /* rm 516 */
  H1P1V10a = add_vertex(H1P1,-5.3,1049.7);
  H1P1V10b = add_vertex(H1P1,-5.3,1013.5);
  H1P1V10c = add_vertex(H1P1,-244.1,1013.5);
  H1P1V10d = add_vertex(H1P1,-244.1,1119.6);
  H1P1V10e = add_vertex(H1P1,-5.3,1119.6);
```

```
H1P1V10f = add_vertex(H1P1,-5.3,1085.4);

H1P1V11 = add_vertex(H1P1,0.0,1085.4);
H1P1V12= add_vertex(H1P1,0.0,1161.7); /* rm 518 */
H1P1V12a = add_vertex(H1P1,-5.3,1161.7);
H1P1V12b = add_vertex(H1P1,-5.3,1125.5);
H1P1V12c = add_vertex(H1P1,-244.1,1125.5);
H1P1V12d = add_vertex(H1P1,-244.1,1231.6);
H1P1V12e = add_vertex(H1P1,-5.3,1231.6);
H1P1V12f = add_vertex(H1P1,-5.3,1197.4);

H1P1V13 = add_vertex(H1P1,0.0,1197.4);
H1P1V14 = add_vertex(H1P1,0.0,1273.4); /* rm 520 */
H1P1V14a = add_vertex(H1P1,-5.3,1273.4);
H1P1V14b = add_vertex(H1P1,-5.3,1237.2);
H1P1V14c = add_vertex(H1P1,-244.1,1237.2);
H1P1V14d = add_vertex(H1P1,-244.1,1343.3);
H1P1V14e = add_vertex(H1P1,-5.3,1343.3);
H1P1V14f = add_vertex(H1P1,-5.3,1309.1);

H1P1V15 = add_vertex(H1P1,0.0,1309.1);
H1P1V16 = add_vertex(H1P1,0.0,1429.6); /* rm 522R */
H1P1V16a = add_vertex(H1P1,-5.3,1429.6);
H1P1V16b = add_vertex(H1P1,-5.3,1393.4);
H1P1V16c = add_vertex(H1P1,-244.1,1393.4);
H1P1V16d = add_vertex(H1P1,-244.1,1499.5);
H1P1V16e = add_vertex(H1P1,-5.3,1499.5);
H1P1V16f = add_vertex(H1P1,-5.3,1461.3);

H1P1V17 = add_vertex(H1P1,0.0,1461.3);
H1P1V18 = add_vertex(H1P1,0.0,1488.0); /* FD #1 */
H1P1V18a = add_vertex(H1P1,-5.5,1488.0);
H1P1V18b = add_vertex(H1P1,-5.5,1486.0);
H1P1V18c = add_vertex(H1P1,-50.0,1486.0);
H1P1V18d = add_vertex(H1P1,-50.0,1562.0);
H1P1V18e = add_vertex(H1P1,-5.5,1562.0);
H1P1V18f = add_vertex(H1P1,-5.5,1560.0);

H1P1V19 = add_vertex(H1P1,0.0,1560.0);
H1P1V20 = add_vertex(H1P1,0.0,1583.3); /* rm 524 */
H1P1V20a = add_vertex(H1P1,-5.3,1583.3);
H1P1V20b = add_vertex(H1P1,-5.3,1547.1);
H1P1V20c = add_vertex(H1P1,-244.1,1547.1);
H1P1V20d = add_vertex(H1P1,-244.1,1653.2);
H1P1V20e = add_vertex(H1P1,-5.3,1653.2);
H1P1V20f = add_vertex(H1P1,-5.3,1619.0);

H1P1V21 = add_vertex(H1P1,0.0,1619.0);
H1P1V22 = add_vertex(H1P1,0.0,1650.4); /* water cooler */
H1P1V22a = add_vertex(H1P1,-30.0,1650.4);
H1P1V22b = add_vertex(H1P1,-30.0,1684.5);
H1P1V23 = add_vertex(H1P1,0.0,1684.5);
H1P1V24 = add_vertex(H1P1,0.0,1754.5); /* rm 526R */
H1P1V24a = add_vertex(H1P1,-5.3,1754.5);
H1P1V24b = add_vertex(H1P1,-5.3,1718.3);
H1P1V24c = add_vertex(H1P1,-244.1,1718.3);
H1P1V24d = add_vertex(H1P1,-244.1,1790.0);
H1P1V24e = add_vertex(H1P1,-5.3,1790.0);
H1P1V24f = add_vertex(H1P1,-5.3,1786.2);

H1P1V25 = add_vertex(H1P1,0.0,1786.2);
H1P1V26 = add_vertex(H1P1,0.0,1836.4); /* rm 528A */
```

```
H1P1V26a = add_vertex(H1P1,-5.3,1836.4);
H1P1V26b = add_vertex(H1P1,-5.3,1800.2);
H1P1V26c = add_vertex(H1P1,-244.1,1800.2);
H1P1V26d = add_vertex(H1P1,-244.1,1875.0);
H1P1V26e = add_vertex(H1P1,-5.3,1875.0);
H1P1V26f = add_vertex(H1P1,-5.3,1872.1);

H1P1V27 = add_vertex(H1P1,0.0,1872.1);
H1P1V28 = add_vertex(H1P1,0.0,1919.1); /* rm 528B */
H1P1V28a = add_vertex(H1P1,-5.3,1919.1);
H1P1V28b = add_vertex(H1P1,-5.3,1882.9);
H1P1V28c = add_vertex(H1P1,-244.1,1882.9);
H1P1V28d = add_vertex(H1P1,-244.1,1989.0);
H1P1V28e = add_vertex(H1P1,-5.3,1989.0);
H1P1V28f = add_vertex(H1P1,-5.3,1954.8);

H1P1V29 = add_vertex(H1P1,0.0,1954.8);
H1P1V30 = add_vertex(H1P1,0.0,2030.4); /* rm 530A */
H1P1V30a = add_vertex(H1P1,-5.3,2030.4);
H1P1V30b = add_vertex(H1P1,-5.3,1994.2);
H1P1V30c = add_vertex(H1P1,-244.1,1994.2);
H1P1V30d = add_vertex(H1P1,-244.1,2100.3);
H1P1V30e = add_vertex(H1P1,-5.3,2100.3);
H1P1V30f = add_vertex(H1P1,-5.3,2066.1);

H1P1V31 = add_vertex(H1P1,0.0,2066.1);
H1P1V32 = add_vertex(H1P1,0.0,2195.1); /* rm 530B */
H1P1V32a = add_vertex(H1P1,-5.3,2195.1);
H1P1V32b = add_vertex(H1P1,-5.3,2158.8);
H1P1V32c = add_vertex(H1P1,-244.1,2158.8);
H1P1V32d = add_vertex(H1P1,-244.1,2250.0);
H1P1V32e = add_vertex(H1P1,-5.3,2250.0);
H1P1V32f = add_vertex(H1P1,-5.3,2230.8);

H1P1V33 = add_vertex(H1P1,0.0,2230.8);
H1P1V34 = add_vertex(H1P1,0.0,2253.8); /* rm 530C */
H1P1V34a = add_vertex(H1P1,-5.3,2253.8);
H1P1V34b = add_vertex(H1P1,-5.3,2251.0);
H1P1V34c = add_vertex(H1P1,-244.1,2251.0);
H1P1V34d = add_vertex(H1P1,-244.1,2350.0);
H1P1V34e = add_vertex(H1P1,-5.3,2350.0);
H1P1V34f = add_vertex(H1P1,-5.3,2289.5);

H1P1V35 = add_vertex(H1P1,0.0,2289.5);
H1P1V36 = add_vertex(H1P1,0.0,2351.2);
H1P1V37 = add_vertex(H1P1,98.0,2351.2);
H1P1V38 = add_vertex(H1P1,98.0,2171.9); /* rm 421 */
H1P1V38a = add_vertex(H1P1,103.3,2171.9);
H1P1V38b = add_vertex(H1P1,103.3,2206.6);
H1P1V38c = add_vertex(H1P1,342.1,2206.6);
H1P1V38d = add_vertex(H1P1,342.1,2099.5);
H1P1V38e = add_vertex(H1P1,103.3,2099.5);
H1P1V38f = add_vertex(H1P1,103.3,2136.2);

H1P1V39 = add_vertex(H1P1,98.0,2136.2);
H1P1V40 = add_vertex(H1P1,98.0,1937.7); /* rm 531 */
H1P1V40a = add_vertex(H1P1,103.3,1937.7);
H1P1V40b = add_vertex(H1P1,103.3,1972.7);
H1P1V40c = add_vertex(H1P1,342.1,1972.7);
H1P1V40d = add_vertex(H1P1,342.1,1865.6);
H1P1V40e = add_vertex(H1P1,103.3,1865.6);
H1P1V40f = add_vertex(H1P1,103.3,1877.7);
```

```
H1P1V41 = add_vertex(H1P1,98.0,1877.7);
H1P1V42 = add_vertex(H1P1,98.0,1744.5); /* rm 529 */
H1P1V42a = add_vertex(H1P1,103.3,1744.5);
H1P1V42b = add_vertex(H1P1,103.3,1779.5);
H1P1V42c = add_vertex(H1P1,342.1,1779.5);
H1P1V42d = add_vertex(H1P1,342.1,1672.4);
H1P1V42e = add_vertex(H1P1,103.3,1672.4);
H1P1V42f = add_vertex(H1P1,103.3,1684.5);

H1P1V43 = add_vertex(H1P1,98.0,1684.5);
H1P1V44 = add_vertex(H1P1,98.0,1522.4); /* rm 527 */
H1P1V44a = add_vertex(H1P1,103.3,1522.4);
H1P1V44b = add_vertex(H1P1,103.3,1557.4);
H1P1V44c = add_vertex(H1P1,342.1,1557.4);
H1P1V44d = add_vertex(H1P1,342.1,1450.3);
H1P1V44e = add_vertex(H1P1,103.3,1450.3);
H1P1V44f = add_vertex(H1P1,103.3,1462.4);

H1P1V45 = add_vertex(H1P1,98.0,1462.4);
H1P1V46 = add_vertex(H1P1,98.0,1342.7); /* rm 525 */
H1P1V46a = add_vertex(H1P1,103.3,1342.7);
H1P1V46b = add_vertex(H1P1,103.3,1377.7);
H1P1V46c = add_vertex(H1P1,342.1,1377.7);
H1P1V46d = add_vertex(H1P1,342.1,1270.6);
H1P1V46e = add_vertex(H1P1,103.3,1270.6);
H1P1V46f = add_vertex(H1P1,103.3,1307.0);

H1P1V47 = add_vertex(H1P1,98.0,1307.0);
H1P1V48 = add_vertex(H1P1,98.0,1118.8); /* rm 523 */
H1P1V48a = add_vertex(H1P1,103.3,1118.8);
H1P1V48b = add_vertex(H1P1,103.3,1153.8);
H1P1V48c = add_vertex(H1P1,342.1,1153.8);
H1P1V48d = add_vertex(H1P1,342.1,1046.7);
H1P1V48e = add_vertex(H1P1,103.3,1046.7);
H1P1V48f = add_vertex(H1P1,103.3,1083.1);

H1P1V49 = add_vertex(H1P1,98.0,1083.1);
H1P1V50 = add_vertex(H1P1,98.0,796.1); /* rm 521 */
H1P1V50a = add_vertex(H1P1,103.3,796.1);
H1P1V50b = add_vertex(H1P1,103.3,831.1);
H1P1V50c = add_vertex(H1P1,342.1,831.1);
H1P1V50d = add_vertex(H1P1,342.1,724.0);
H1P1V50e = add_vertex(H1P1,103.3,724.0);
H1P1V50f = add_vertex(H1P1,103.3,760.4);

H1P1V51 = add_vertex(H1P1,98.0,760.4);
H1P1V52 = add_vertex(H1P1,98.0,564.5); /* rm 519 */
H1P1V52a = add_vertex(H1P1,103.3,564.5);
H1P1V52b = add_vertex(H1P1,103.3,599.5);
H1P1V52c = add_vertex(H1P1,342.1,599.5);
H1P1V52d = add_vertex(H1P1,342.1,492.4);
H1P1V52e = add_vertex(H1P1,103.3,492.4);
H1P1V52f = add_vertex(H1P1,103.3,528.8);

H1P1V53 = add_vertex(H1P1,98.0,528.8);
H1P1V54 = add_vertex(H1P1,98.0,413.9);  /* corners */
H1P1V55 = add_vertex(H1P1,257.9,413.9); /* rm ? */
H1P1V55a = add_vertex(H1P1,257.9,419.2);
H1P1V55b = add_vertex(H1P1,221.7,419.2);
H1P1V55c = add_vertex(H1P1,221.7,500.0);
H1P1V55d = add_vertex(H1P1,300.0,500.0);
H1P1V55e = add_vertex(H1P1,300.0,419.2);
```

```
H1P1V55f = add_vertex(H1P1,293.9,419.2);

H1P1V56 = add_vertex(H1P1,293.9,413.9);
H1P1V57 = add_vertex(H1P1,337.5,413.9);
H1P1V58 = add_vertex(H1P1,337.5,402.6); /* office */
H1P1V58a = add_vertex(H1P1,342.8,402.6);
H1P1V58b = add_vertex(H1P1,342.8,600.0);
H1P1V58c = add_vertex(H1P1,449.9,600.0);
H1P1V58d = add_vertex(H1P1,449.9,330.0);
H1P1V58e = add_vertex(H1P1,342.8,330.0);
H1P1V58f = add_vertex(H1P1,342.8,342.6);

H1P1V59 = add_vertex(H1P1,337.5,342.6);
H1P1V60 = add_vertex(H1P1,337.5,310.2); /* rm 511 */
H1P1V60a = add_vertex(H1P1,342.8,310.2);
H1P1V60b = add_vertex(H1P1,342.8,315.0);
H1P1V60c = add_vertex(H1P1,449.9,315.0);
H1P1V60d = add_vertex(H1P1,449.9,0.0);
H1P1V60e = add_vertex(H1P1,342.8,0.0);
H1P1V60f = add_vertex(H1P1,342.8,274.5);

H1P1V61 = add_vertex(H1P1,337.5,274.5);
H1P1V62 = add_vertex(H1P1,337.5,267.4);
H1P1V63 = add_vertex(H1P1,306.9,267.4); /* elev 1 (left)*/
H1P1V63a = add_vertex(H1P1,306.9,267.7);
H1P1V63b = add_vertex(H1P1,303.9,267.7);
H1P1V63c = add_vertex(H1P1,303.9,255.7);
H1P1V63d = add_vertex(H1P1,277.9,255.7);
H1P1V63e = add_vertex(H1P1,251.9,255.7);
H1P1V63f = add_vertex(H1P1,251.9,267.7);
H1P1V63g = add_vertex(H1P1,248.9,267.7);
H1P1V64 = add_vertex(H1P1,248.9,267.4);
H1P1V65 = add_vertex(H1P1,192.2,267.4);

                                                    /* elev 2 */
H1P1V65a = add_vertex(H1P1,192.2,267.7);
H1P1V65b = add_vertex(H1P1,189.2,267.7);
H1P1V65c = add_vertex(H1P1,189.2,255.7);
H1P1V65d = add_vertex(H1P1,163.2,255.7);
H1P1V65e = add_vertex(H1P1,137.2,255.7);
H1P1V65f = add_vertex(H1P1,137.2,267.7);
H1P1V65g = add_vertex(H1P1,134.2,267.7);
H1P1V66 = add_vertex(H1P1,134.2,267.4);

H1P1V67 = add_vertex(H1P1,98.0,267.4);
H1P1V68 = add_vertex(H1P1,98.0,100.0); /* stairwell */
H1P1V68a = add_vertex(H1P1,103.3,100.0);
H1P1V68b = add_vertex(H1P1,103.3,125.0);
H1P1V68c = add_vertex(H1P1,150.0,125.0);
H1P1V68d = add_vertex(H1P1,150.0,40.0);
H1P1V68e = add_vertex(H1P1,103.3,40.0);
H1P1V68f = add_vertex(H1P1,103.3,64.3);
H1P1V69 = add_vertex(H1P1,98.0,64.3);
H1P1V70 = add_vertex(H1P1,98.0,0.0);

H1P2=add_pg(H1,102.0,0.1); /*main ceiling*/
H1P2V1 = add_vertex(H1P2,0.0,0.0);
H1P2V2 = add_vertex(H1P2,0.0,2351.2);
H1P2V3 = add_vertex(H1P2,98.0,2351.2);
H1P2V4 = add_vertex(H1P2,98.0,0.0);

H1P3=add_pg(H1,113.3,0.1); /*elev ceiling*/
```

```
H1P3V1 = add_vertex(H1P3,98.0,267.4);
H1P3V2 = add_vertex(H1P3,98.0,413.9);
H1P3V3 = add_vertex(H1P3,337.5,413.9);
H1P3V4 = add_vertex(H1P3,337.5,267.4);

H1P4 = add_pg(H1,84.0,0,1);  /*rm 506 door jam ceiling*/
H1P4V1= add_vertex(H1P4,0.0,239.5);
H1P4V2= add_vertex(H1P4,-5.3,239.5);
H1P4V3= add_vertex(H1P4,-5.3,275.2);
H1P4V4= add_vertex(H1P4,0.0,275.2);
H1P5 = add_pg(H1,84.0,0,1);  /*rm 510 door jam ceiling*/
H1P5V1= add_vertex(H1P5,0.0,713.7);
H1P5V2= add_vertex(H1P5,-5.3,713.7);
H1P5V3= add_vertex(H1P5,-5.3,749.4);
H1P5V4= add_vertex(H1P5,0.0,749.4);
H1P6 = add_pg(H1,84.0,0,1);  /*rm 512 door jam ceiling*/
H1P6V1= add_vertex(H1P6,0.0,825.9);
H1P6V2= add_vertex(H1P6,-5.3,825.9);
H1P6V3= add_vertex(H1P6,-5.3,861.6);
H1P6V4= add_vertex(H1P6,0.0,861.6);
H1P7 = add_pg(H1,84.0,0,1);  /*rm 514 door jam ceiling*/
H1P7V1= add_vertex(H1P7,0.0,937.5);
H1P7V2= add_vertex(H1P7,-5.3,937.5);
H1P7V3= add_vertex(H1P7,-5.3,973.2);
H1P7V4= add_vertex(H1P7,0.0,973.2);
H1P8 = add_pg(H1,84.0,0,1);  /*rm 516 door jam ceiling*/
H1P8V1= add_vertex(H1P8,0.0,1049.7);
H1P8V2= add_vertex(H1P8,-5.3,1049.7);
H1P8V3= add_vertex(H1P8,-5.3,1085.4);
H1P8V4= add_vertex(H1P8,0.0,1085.4);
H1P9 = add_pg(H1,84.0,0,1);  /*rm 518 door jam ceiling*/
H1P9V1= add_vertex(H1P9,0.0,1161.7);
H1P9V2= add_vertex(H1P9,-5.3,1161.7);
H1P9V3= add_vertex(H1P9,-5.3,1197.4);
H1P9V4= add_vertex(H1P9,0.0,1197.4);
H1P10 = add_pg(H1,84.0,0,1);  /*rm 520 door jam ceiling*/
H1P10V1= add_vertex(H1P10,0.0,1273.4);
H1P10V2= add_vertex(H1P10,-5.3,1273.4);
H1P10V3= add_vertex(H1P10,-5.3,1309.1);
H1P10V4= add_vertex(H1P10,0.0,1309.1);
H1P11 = add_pg(H1,84.0,0,1);  /*rm 522R door jam ceiling*/
H1P11V1= add_vertex(H1P11,0.0,1429.6);
H1P11V2= add_vertex(H1P11,-5.3,1429.6);
H1P11V3= add_vertex(H1P11,-5.3,1461.3);
H1P11V4= add_vertex(H1P11,0.0,1461.3);
H1P12 = add_pg(H1,84.0,0,1);  /*rm FD 1 door jam ceiling*/
H1P12V1= add_vertex(H1P12,0.0,1488.0);
H1P12V2= add_vertex(H1P12,-5.5,1488.0);
H1P12V3= add_vertex(H1P12,-5.5,1560.0);
H1P12V4= add_vertex(H1P12,0.0,1560.0);
H1P13 = add_pg(H1,84.0,0,1);  /*rm 524 door jam ceiling*/
H1P13V1= add_vertex(H1P13,0.0,1583.3);
H1P13V2= add_vertex(H1P13,-5.3,1583.3);
H1P13V3= add_vertex(H1P13,-5.3,1619.0);
H1P13V4= add_vertex(H1P13,0.0,1619.0);
H1P14 = add_pg(H1,84.0,0,1);  /* 526R ceiling*/
H1P14V1= add_vertex(H1P14,0.0,1754.5);
H1P14V2= add_vertex(H1P14,-5.3,1754.5);
H1P14V3= add_vertex(H1P14,-5.3,1786.2);
H1P14V4= add_vertex(H1P14,0.0,1786.2);
H1P15 = add_pg(H1,84.0,0,1);  /*rm 528A door jam ceiling*/
H1P15V1= add_vertex(H1P15,0.0,1836.4);
```

```
H1P15V2 = add_vertex(H1P15,-5.3,1836.4);
H1P15V3 = add_vertex(H1P15,-5.3,1872.1);
H1P15V4 = add_vertex(H1P15,0.0,1872.1);
H1P16 = add_pg(H1,84.0,0,1);  /*rm 528B door jam ceiling*/
H1P16V1 = add_vertex(H1P16,0.0,1919.1);
H1P16V2 = add_vertex(H1P16,-5.3,1919.1);
H1P16V3 = add_vertex(H1P16,-5.3,1954.8);
H1P16V4 = add_vertex(H1P16,0.0,1954.8);
H1P17 = add_pg(H1,84.0,0,1);  /*rm 530A door jam ceiling*/
H1P17V1 = add_vertex(H1P17,0.0,2030.4);
H1P17V2 = add_vertex(H1P17,-5.3,2030.4);
H1P17V3 = add_vertex(H1P17,-5.3,2066.1);
H1P17V4 = add_vertex(H1P17,0.0,2066.1);
H1P18 = add_pg(H1,84.0,0,1);  /*rm 530B door jam ceiling*/
H1P18V1 = add_vertex(H1P18,0.0,2195.1);
H1P18V2 = add_vertex(H1P18,-5.3,2195.1);
H1P18V3 = add_vertex(H1P18,-5.3,2230.8);
H1P18V4 = add_vertex(H1P18,0.0,2230.8);
H1P19 = add_pg(H1,84.0,0,1);  /*rm 530C door jam ceiling*/
H1P19V1 = add_vertex(H1P19,0.0,2253.8);
H1P19V2 = add_vertex(H1P19,-5.3,2253.8);
H1P19V3 = add_vertex(H1P19,-5.3,2289.5);
H1P19V4 = add_vertex(H1P19,0.0,2289.5);
H1P20 = add_pg(H1,84.0,0,1);  /*rm 421 door jam ceiling*/
H1P20V1 = add_vertex(H1P20,98.0,2171.9);
H1P20V2 = add_vertex(H1P20,103.3,2171.9);
H1P20V3 = add_vertex(H1P20,103.3,2136.2);
H1P20V4 = add_vertex(H1P20,98.0,2136.2);
H1P21 = add_pg(H1,84.0,0,1);  /*rm 531 door jam ceiling*/
H1P21V1 = add_vertex(H1P21,98.0,1937.7);
H1P21V2 = add_vertex(H1P21,103.3,1937.7);
H1P21V3 = add_vertex(H1P21,103.3,1877.7);
H1P21V4 = add_vertex(H1P21,98.0,1877.7);
H1P22 = add_pg(H1,84.0,0,1);  /*rm 529 door jam ceiling*/
H1P22V1 = add_vertex(H1P22,98.0,1744.5);
H1P22V2 = add_vertex(H1P22,103.3,1744.5);
H1P22V3 = add_vertex(H1P22,103.3,1684.5);
H1P22V4 = add_vertex(H1P22,98.0,1684.5);
H1P23 = add_pg(H1,84.0,0,1);  /*rm 527 door jam ceiling*/
H1P23V1 = add_vertex(H1P23,98.0,1522.4);
H1P23V2 = add_vertex(H1P23,103.3,1522.4);
H1P23V3 = add_vertex(H1P23,103.3,1462.4);
H1P23V4 = add_vertex(H1P23,98.0,1462.4);
H1P24 = add_pg(H1,84.0,0,1);  /*rm 525 door jam ceiling*/
H1P24V1 = add_vertex(H1P24,98.0,1342.7);
H1P24V2 = add_vertex(H1P24,103.3,1342.7);
H1P24V3 = add_vertex(H1P24,103.0,1307.0);
H1P24V4 = add_vertex(H1P24,98.0,1307.0);
H1P25 = add_pg(H1,84.0,0,1);  /*rm 523 door jam ceiling*/
H1P25V1 = add_vertex(H1P25,98.0,1118.8);
H1P25V2 = add_vertex(H1P25,103.3,1118.8);
H1P25V3 = add_vertex(H1P25,103.3,1083.1);
H1P25V4 = add_vertex(H1P25,98.0,1083.1);
H1P26 = add_pg(H1,84.0,0,1);  /*rm 521 door jam ceiling*/
H1P26V1 = add_vertex(H1P26,98.0,796.1);
H1P26V2 = add_vertex(H1P26,103.3,796.1);
H1P26V3 = add_vertex(H1P26,103.3,760.4);
H1P26V4 = add_vertex(H1P26,98.0,760.4);
H1P27 = add_pg(H1,84.0,0,1);  /*rm 519 door jam ceiling*/
H1P27V1 = add_vertex(H1P27,98.0,564.5);
H1P27V2 = add_vertex(H1P27,103.3,564.5);
H1P27V3 = add_vertex(H1P27,103.3,528.8);
```

```
H1P27V4= add_vertex(H1P27,98.0,528.8);
H1P28 = add_pg(H1,84.0,0,1);  /*rm ? door jam ceiling*/
H1P28V1= add_vertex(H1P28,257.9,413.9);
H1P28V2= add_vertex(H1P28,257.9,419.2);
H1P28V3= add_vertex(H1P28,293.9,419.2);
H1P28V4= add_vertex(H1P28,293.9,413.9);
H1P29 = add_pg(H1,84.0,0,1);  /* office door jam ceiling*/
H1P29V1= add_vertex(H1P29,337.5,402.6);
H1P29V2= add_vertex(H1P29,342.8,402.6);
H1P29V3= add_vertex(H1P29,342.8,342.6);
H1P29V4= add_vertex(H1P29,337.5,342.6);
H1P30 = add_pg(H1,84.0,0,1);  /* rm 511 door jam ceiling*/
H1P30V1= add_vertex(H1P30,337.5,310.2);
H1P30V2= add_vertex(H1P30,342.8,310.2);
H1P30V3= add_vertex(H1P30,342.8,274.5);
H1P30V4= add_vertex(H1P30,337.5,274.5);

H1P31 = add_pg(H1,83.8,0,1);  /* elev 1 door jam ceiling*/
H1P31V1= add_vertex(H1P31,303.9,267.7);
H1P31V2= add_vertex(H1P31,303.9,255.7);
H1P31V3= add_vertex(H1P31,277.9,255.7);
H1P31V4= add_vertex(H1P31,251.9,255.7);
H1P31V5= add_vertex(H1P31,251.9,267.7);
H1P32 = add_pg(H1,83.8,0,1);  /* elev 2 door jam ceiling*/
H1P32V1= add_vertex(H1P32,189.2,267.7);
H1P32V2= add_vertex(H1P32,189.2,255.7);
H1P32V3= add_vertex(H1P32,163.2,255.7);
H1P32V4= add_vertex(H1P32,137.2,255.7);
H1P32V5= add_vertex(H1P32,137.2,267.7);

H1P63 = add_pg(H1,86.8,0,1);  /* elev 1 ceiling*/
H1P63V1= add_vertex(H1P63,306.9,267.4);
H1P63V2= add_vertex(H1P63,306.9,267.7);
H1P63V3= add_vertex(H1P63,248.9,267.7);
H1P63V4= add_vertex(H1P63,248.9,267.4);


H1P64 = add_pg(H1,86.8,0,1);  /* elev 2 ceiling*/
H1P64V1= add_vertex(H1P64,192.2,267.4);
H1P64V2= add_vertex(H1P64,192.2,267.7);
H1P64V3= add_vertex(H1P64,134.2,267.7);
H1P64V4= add_vertex(H1P64,134.2,267.4);

H1P33 = add_pg(H1,84.0,0,1);  /* stairwell door jam ceiling*/
H1P33V1= add_vertex(H1P33,98.0,100.0);
H1P33V2= add_vertex(H1P33,103.3,100.0);
H1P33V3= add_vertex(H1P33,103.3,64.3);
H1P33V4= add_vertex(H1P33,98.0,64.3);

H1P34 = add_pg(H1,144.0,0,1);  /*rm 506 ceiling*/
H1P34V1= add_vertex(H1P34,-5.3,203.3);
H1P34V2= add_vertex(H1P34,-244.1,203.3);
H1P34V3= add_vertex(H1P34,-244.1,309.4);
H1P34V4= add_vertex(H1P34,-5.3,309.4);
H1P35 = add_pg(H1,144.0,0,1);  /*rm 510 ceiling*/
H1P35V1= add_vertex(H1P35,-5.3,677.5);
H1P35V2= add_vertex(H1P35,-244.1,677.5);
H1P35V3= add_vertex(H1P35,-244.1,783.6);
H1P35V4= add_vertex(H1P35,-5.3,783.6);
H1P36 = add_pg(H1,144.0,0,1);  /*rm 512 ceiling*/
H1P36V1= add_vertex(H1P36,-5.3,789.7);
H1P36V2= add_vertex(H1P36,-244.1,789.7);
```

```
H1P36V3= add_vertex(H1P36,-244.1,895.8);
H1P36V4= add_vertex(H1P36,-5.3,895.8);
H1P37 = add_pg(H1,144.0,0,1);  /*rm 514 ceiling*/
H1P37V1= add_vertex(H1P37,-5.3,901.3);
H1P37V2= add_vertex(H1P37,-244.1,901.3);
H1P37V3= add_vertex(H1P37,-244.1,1007.4);
H1P37V4= add_vertex(H1P37,-5.3,1007.4);
H1P38 = add_pg(H1,144.0,0,1);  /*rm 516 ceiling*/
H1P38V1= add_vertex(H1P38,-5.3,1013.5);
H1P38V2= add_vertex(H1P38,-244.1,1013.5);
H1P38V3= add_vertex(H1P38,-244.1,1119.6);
H1P38V4= add_vertex(H1P38,-5.3,1119.6);
H1P39 = add_pg(H1,144.0,0,1);  /*rm 518 ceiling*/
H1P39V1= add_vertex(H1P39,-5.3,1125.5);
H1P39V2= add_vertex(H1P39,-244.1,1125.5);
H1P39V3= add_vertex(H1P39,-244.1,1231.6);
H1P39V4= add_vertex(H1P39,-5.3,1231.6);
H1P40 = add_pg(H1,144.0,0,1);  /*rm 520 ceiling*/
H1P40V1= add_vertex(H1P40,-5.3,1237.2);
H1P40V2= add_vertex(H1P40,-244.1,1237.2);
H1P40V3= add_vertex(H1P40,-244.1,1343.3);
H1P40V4= add_vertex(H1P40,-5.3,1343.3);
H1P41 = add_pg(H1,144.0,0,1);  /*rm 522R ceiling*/
H1P41V1= add_vertex(H1P41,-5.3,1393.4);
H1P41V2= add_vertex(H1P41,-244.1,1393.4);
H1P41V3= add_vertex(H1P41,-244.1,1499.5);
H1P41V4= add_vertex(H1P41,-5.3,1499.5);
H1P42 = add_pg(H1,144.0,0,1);  /* FD1 ceiling*/
H1P42V1= add_vertex(H1P42,-5.5,1486.0);
H1P42V2= add_vertex(H1P42,-50.0,1486.0);
H1P42V3= add_vertex(H1P42,-50.0,1562.0);
H1P42V4= add_vertex(H1P42,-5.5,1562.0);
H1P43 = add_pg(H1,144.0,0,1);  /*rm 524 ceiling*/
H1P43V1= add_vertex(H1P43,-5.3,1547.1);
H1P43V2= add_vertex(H1P43,-244.1,1547.1);
H1P43V3= add_vertex(H1P43,-244.1,1653.2);
H1P43V4= add_vertex(H1P43,-5.3,1653.2);
H1P44 = add_pg(H1,84.0,0,1);  /* water fountain ceiling*/
H1P44V1= add_vertex(H1P44,0.0,1650.4);
H1P44V2= add_vertex(H1P44,-30.0,1650.4);
H1P44V3= add_vertex(H1P44,-30.0,1684.5);
H1P44V4= add_vertex(H1P44,0.0,1684.5);
H1P45 = add_pg(H1,144.0,0,1);  /*rm 526R ceiling*/
H1P45V1= add_vertex(H1P45,-5.3,1718.3);
H1P45V2= add_vertex(H1P45,-244.1,1718.3);
H1P45V3= add_vertex(H1P45,-244.1,1790.0);
H1P45V4= add_vertex(H1P45,-5.3,1790.0);
H1P46 = add_pg(H1,144.0,0,1);  /*rm 528A ceiling*/
H1P46V1= add_vertex(H1P46,-5.3,1800.2);
H1P46V2= add_vertex(H1P46,-244.1,1800.2);
H1P46V3= add_vertex(H1P46,-244.1,1875.0);
H1P46V4= add_vertex(H1P46,-5.3,1875.0);
H1P47 = add_pg(H1,144.0,0,1);  /*rm 528B ceiling*/
H1P47V1= add_vertex(H1P47,-5.3,1882.9);
H1P47V2= add_vertex(H1P47,-244.1,1882.9);
H1P47V3= add_vertex(H1P47,-244.1,1989.0);
H1P47V4= add_vertex(H1P47,-5.3,1989.0);
H1P48 = add_pg(H1,144.0,0,1);  /*rm 530A ceiling*/
H1P48V1= add_vertex(H1P48,-5.3,1994.2);
H1P48V2= add_vertex(H1P48,-244.1,1994.2);
H1P48V3= add_vertex(H1P48,-244.1,2100.3);
H1P48V4= add_vertex(H1P48,-5.3,2100.3);
```

```
H1P49 = add_pg(H1,144.0,0,1);  /*rm 530B ceiling*/
H1P49V1= add_vertex(H1P49,-5.3,2158.8);
H1P49V2= add_vertex(H1P49,-244.1,2158.8);
H1P49V3= add_vertex(H1P49,-244.1,2250.0);
H1P49V4= add_vertex(H1P49,-5.3,2250.0);
H1P50 = add_pg(H1,144.0,0,1);  /*rm 530C ceiling*/
H1P50V1= add_vertex(H1P50,-5.3,2251.0);
H1P50V2= add_vertex(H1P50,-244.1,2251.0);
H1P50V3= add_vertex(H1P50,-244.1,2350.0);
H1P50V4= add_vertex(H1P50,-5.3,2350.0);

/* following ceilings are incorrect based on 35" to either side of door*/

H1P51 = add_pg(H1,144.0,0,1);  /*rm 421 ceiling*/
H1P51V1= add_vertex(H1P51,103.3,2206.6);
H1P51V2= add_vertex(H1P51,342.1,2206.6);
H1P51V3= add_vertex(H1P51,342.1,2099.5);
H1P51V4= add_vertex(H1P51,103.3,2099.5);
H1P52 = add_pg(H1,144.0,0,1);  /*rm 531 ceiling*/
H1P52V1= add_vertex(H1P52,103.3,1972.7);
H1P52V2= add_vertex(H1P52,342.1,1972.7);
H1P52V3= add_vertex(H1P52,342.1,1865.6);
H1P52V4= add_vertex(H1P52,103.3,1865.6);
H1P53 = add_pg(H1,144.0,0,1);  /*rm 529 ceiling*/
H1P53V1= add_vertex(H1P53,103.3,1779.5);
H1P53V2= add_vertex(H1P53,342.1,1779.5);
H1P53V3= add_vertex(H1P53,342.1,1672.4);
H1P53V4= add_vertex(H1P53,103.3,1672.4);
H1P54 = add_pg(H1,144.0,0,1);  /*rm 527 ceiling*/
H1P54V1= add_vertex(H1P54,103.3,1557.4);
H1P54V2= add_vertex(H1P54,342.1,1557.4);
H1P54V3= add_vertex(H1P54,342.1,1450.3);
H1P54V4= add_vertex(H1P54,103.3,1450.3);
H1P55 = add_pg(H1,144.0,0,1);  /*rm 525 ceiling*/
H1P55V1= add_vertex(H1P55,103.3,1377.7);
H1P55V2= add_vertex(H1P55,342.1,1377.7);
H1P55V3= add_vertex(H1P55,342.1,1270.6);
H1P55V4= add_vertex(H1P55,103.3,1270.6);
H1P56 = add_pg(H1,144.0,0,1);  /*rm 523 ceiling*/
H1P56V1= add_vertex(H1P56,103.3,1153.8);
H1P56V2= add_vertex(H1P56,342.1,1153.8);
H1P56V3= add_vertex(H1P56,342.1,1046.7);
H1P56V4= add_vertex(H1P56,103.3,1046.7);
H1P57 = add_pg(H1,144.0,0,1);  /*rm 521 ceiling*/
H1P57V1= add_vertex(H1P57,103.3,831.1);
H1P57V2= add_vertex(H1P57,342.1,831.1);
H1P57V3= add_vertex(H1P57,342.1,724.0);
H1P57V4= add_vertex(H1P57,103.3,724.0);
H1P58 = add_pg(H1,144.0,0,1);  /*rm 519 ceiling*/
H1P58V1= add_vertex(H1P58,103.3,599.5);
H1P58V2= add_vertex(H1P58,342.1,599.5);
H1P58V3= add_vertex(H1P58,342.1,492.4);
H1P58V4= add_vertex(H1P58,103.3,492.4);
H1P59 = add_pg(H1,144.0,0,1);  /*rm ? ceiling*/
H1P59V1= add_vertex(H1P59,221.7,419.2);
H1P59V2= add_vertex(H1P59,221.7,500.0);
H1P59V3= add_vertex(H1P59,300.0,500.0);
H1P59V4= add_vertex(H1P59,300.0,419.2);
H1P60 = add_pg(H1,144.0,0,1);  /* office ceiling*/
H1P60V1= add_vertex(H1P60,342.8,600.0);
H1P60V2= add_vertex(H1P60,449.9,600.0);
H1P60V3= add_vertex(H1P60,449.9,330.0);
```

```
H1P60V4= add_vertex(H1P60,342.8,330.0);
H1P61 = add_pg(H1,144.0,0.1);  /*rm 511 ceiling*/
H1P61V1= add_vertex(H1P61,342.8,315.0);
H1P61V2= add_vertex(H1P61,449.9,315.0);
H1P61V3= add_vertex(H1P61,449.9,0.0);
H1P61V4= add_vertex(H1P61,342.8,0.0);
H1P62 = add_pg(H1,144.0,0.1);  /*rm stairwell ceiling*/
H1P62V1= add_vertex(H1P62,103.3,125.0);
H1P62V2= add_vertex(H1P62,150.0,125.0);
H1P62V3= add_vertex(H1P62,150.0,40.0);
H1P62V4= add_vertex(H1P62,103.3,40.0);

/* Don't forget to add the ceiling associations or else we can't tell
   how high each section of the hallway is*/

add_ceiling(H1P1,H1P2);
add_ceiling(H1P1,H1P3);
add_ceiling(H1P1,H1P4);
add_ceiling(H1P1,H1P5);
add_ceiling(H1P1,H1P6);
add_ceiling(H1P1,H1P7);
add_ceiling(H1P1,H1P8);
add_ceiling(H1P1,H1P9);
add_ceiling(H1P1,H1P10);
add_ceiling(H1P1,H1P11);
add_ceiling(H1P1,H1P12);
add_ceiling(H1P1,H1P13);
add_ceiling(H1P1,H1P14);
add_ceiling(H1P1,H1P15);
add_ceiling(H1P1,H1P16);
add_ceiling(H1P1,H1P17);
add_ceiling(H1P1,H1P18);
add_ceiling(H1P1,H1P19);
add_ceiling(H1P1,H1P20);
add_ceiling(H1P1,H1P21);
add_ceiling(H1P1,H1P22);
add_ceiling(H1P1,H1P23);
add_ceiling(H1P1,H1P24);
add_ceiling(H1P1,H1P25);
add_ceiling(H1P1,H1P26);
add_ceiling(H1P1,H1P27);
add_ceiling(H1P1,H1P28);
add_ceiling(H1P1,H1P29);
add_ceiling(H1P1,H1P30);
add_ceiling(H1P1,H1P31);
add_ceiling(H1P1,H1P32);
add_ceiling(H1P1,H1P33);
add_ceiling(H1P1,H1P34);
add_ceiling(H1P1,H1P35);
add_ceiling(H1P1,H1P36);
add_ceiling(H1P1,H1P37);
add_ceiling(H1P1,H1P38);
add_ceiling(H1P1,H1P39);
add_ceiling(H1P1,H1P40);
add_ceiling(H1P1,H1P41);
add_ceiling(H1P1,H1P42);
add_ceiling(H1P1,H1P43);
add_ceiling(H1P1,H1P44);
add_ceiling(H1P1,H1P45);
add_ceiling(H1P1,H1P46);
add_ceiling(H1P1,H1P47);
add_ceiling(H1P1,H1P48);
```

```
  add_ceiling(H1P1,H1P49);
  add_ceiling(H1P1,H1P50);
  add_ceiling(H1P1,H1P51);
  add_ceiling(H1P1,H1P52);
  add_ceiling(H1P1,H1P53);
  add_ceiling(H1P1,H1P54);
  add_ceiling(H1P1,H1P55);
  add_ceiling(H1P1,H1P56);
  add_ceiling(H1P1,H1P57);
  add_ceiling(H1P1,H1P58);
  add_ceiling(H1P1,H1P59);
  add_ceiling(H1P1,H1P60);
  add_ceiling(H1P1,H1P61);
  add_ceiling(H1P1,H1P62);
  add_ceiling(H1P1,H1P63);
  add_ceiling(H1P1,H1P64);

/* Vertical edges must alway be explicitly added */

add_edge(H1P1V1,H1P2V1);
add_edge(H1P1V2,H1P4V1);  /*link up vert edges of room 506*/
add_edge(H1P1V2a,H1P4V2);
add_edge(H1P1V2b,H1P34V1);
add_edge(H1P1V2c,H1P34V2);
add_edge(H1P1V2d,H1P34V3);
add_edge(H1P1V2e,H1P34V4);
add_edge(H1P1V2f,H1P4V3);
add_edge(H1P1V3,H1P4V4);
add_edge(H1P1V4,H1P5V1);  /*link up vert edges of room 510*/
add_edge(H1P1V4a,H1P5V2);
add_edge(H1P1V4b,H1P35V1);
add_edge(H1P1V4c,H1P35V2);
add_edge(H1P1V4d,H1P35V3);
add_edge(H1P1V4e,H1P35V4);
add_edge(H1P1V4f,H1P5V3);
add_edge(H1P1V5,H1P5V4);
add_edge(H1P1V6,H1P6V1);  /*link up vert edges of room 512 */
add_edge(H1P1V6a,H1P6V2);
add_edge(H1P1V6b,H1P36V1);
add_edge(H1P1V6c,H1P36V2);
add_edge(H1P1V6d,H1P36V3);
add_edge(H1P1V6e,H1P36V4);
add_edge(H1P1V6f,H1P6V3);
add_edge(H1P1V7,H1P6V4);
add_edge(H1P1V8,H1P7V1);  /*link up vert edges of room 514*/
add_edge(H1P1V8a,H1P7V2);
add_edge(H1P1V8b,H1P37V1);
add_edge(H1P1V8c,H1P37V2);
add_edge(H1P1V8d,H1P37V3);
add_edge(H1P1V8e,H1P37V4);
add_edge(H1P1V8f,H1P7V3);
add_edge(H1P1V9,H1P7V4);
add_edge(H1P1V10,H1P8V1);  /*link up vert edges of room 516*/
add_edge(H1P1V10a,H1P8V2);
add_edge(H1P1V10b,H1P38V1);
add_edge(H1P1V10c,H1P38V2);
add_edge(H1P1V10d,H1P38V3);
add_edge(H1P1V10e,H1P38V4);
add_edge(H1P1V10f,H1P8V3);
add_edge(H1P1V11,H1P8V4);
add_edge(H1P1V12,H1P9V1);  /*link up vert edges of room 518*/
add_edge(H1P1V12a,H1P9V2);
```

```
add_edge(H1P1V12b,H1P39V1);
add_edge(H1P1V12c,H1P39V2);
add_edge(H1P1V12d,H1P39V3);
add_edge(H1P1V12e,H1P39V4);
add_edge(H1P1V12f,H1P9V3);
add_edge(H1P1V13,H1P9V4);
add_edge(H1P1V14,H1P10V1);  /*link up vert edges of room 520*/
add_edge(H1P1V14a,H1P10V2);
add_edge(H1P1V14b,H1P40V1);
add_edge(H1P1V14c,H1P40V2);
add_edge(H1P1V14d,H1P40V3);
add_edge(H1P1V14e,H1P40V4);
add_edge(H1P1V14f,H1P10V3);
add_edge(H1P1V15,H1P10V4);
add_edge(H1P1V16,H1P11V1);  /*link up vert edges of room 522R*/
add_edge(H1P1V16a,H1P11V2);
add_edge(H1P1V16b,H1P41V1);
add_edge(H1P1V16c,H1P41V2);
add_edge(H1P1V16d,H1P41V3);
add_edge(H1P1V16e,H1P41V4);
add_edge(H1P1V16f,H1P11V3);
add_edge(H1P1V17,H1P11V4);
add_edge(H1P1V18,H1P12V1);  /*link up vert edges of room FD1*/
add_edge(H1P1V18a,H1P12V2);
add_edge(H1P1V18b,H1P42V1);
add_edge(H1P1V18c,H1P42V2);
add_edge(H1P1V18d,H1P42V3);
add_edge(H1P1V18e,H1P42V4);
add_edge(H1P1V18f,H1P12V3);
add_edge(H1P1V19,H1P12V4);
add_edge(H1P1V20,H1P13V1);  /*link up vert edges of room 524*/
add_edge(H1P1V20a,H1P13V2);
add_edge(H1P1V20b,H1P43V1);
add_edge(H1P1V20c,H1P43V2);
add_edge(H1P1V20d,H1P43V3);
add_edge(H1P1V20e,H1P43V4);
add_edge(H1P1V20f,H1P13V3);
add_edge(H1P1V21,H1P13V4);
add_edge(H1P1V22,H1P44V1);  /*link up vert edges of water fountain*/
add_edge(H1P1V22a,H1P44V2);
add_edge(H1P1V22b,H1P44V3);
add_edge(H1P1V23,H1P44V4);
add_edge(H1P1V24,H1P14V1);  /*link up vert edges of room 526R*/
add_edge(H1P1V24a,H1P14V2);
add_edge(H1P1V24b,H1P45V1);
add_edge(H1P1V24c,H1P45V2);
add_edge(H1P1V24d,H1P45V3);
add_edge(H1P1V24e,H1P45V4);
add_edge(H1P1V24f,H1P14V3);
add_edge(H1P1V25,H1P14V4);
add_edge(H1P1V26,H1P15V1);  /*link up vert edges of room 528A*/
add_edge(H1P1V26a,H1P15V2);
add_edge(H1P1V26b,H1P46V1);
add_edge(H1P1V26c,H1P46V2);
add_edge(H1P1V26d,H1P46V3);
add_edge(H1P1V26e,H1P46V4);
add_edge(H1P1V26f,H1P15V3);
add_edge(H1P1V27,H1P15V4);
add_edge(H1P1V28,H1P16V1);  /*link up vert edges of room 528B*/
add_edge(H1P1V28a,H1P16V2);
add_edge(H1P1V28b,H1P47V1);
add_edge(H1P1V28c,H1P47V2);
```

```
add_edge(H1P1V28d,H1P47V3);
add_edge(H1P1V28e,H1P47V4);
add_edge(H1P1V28f,H1P16V3);
add_edge(H1P1V29,H1P16V4);
add_edge(H1P1V30,H1P17V1);  /*link up vert edges of room 530A*/
add_edge(H1P1V30a,H1P17V2);
add_edge(H1P1V30b,H1P48V1);
add_edge(H1P1V30c,H1P48V2);
add_edge(H1P1V30d,H1P48V3);
add_edge(H1P1V30e,H1P48V4);
add_edge(H1P1V30f,H1P17V3);
add_edge(H1P1V31,H1P17V4);
add_edge(H1P1V32,H1P18V1);  /*link up vert edges of room 530B*/
add_edge(H1P1V32a,H1P18V2);
add_edge(H1P1V32b,H1P49V1);
add_edge(H1P1V32c,H1P49V2);
add_edge(H1P1V32d,H1P49V3);
add_edge(H1P1V32e,H1P49V4);
add_edge(H1P1V32f,H1P18V3);
add_edge(H1P1V33,H1P18V4);
add_edge(H1P1V34,H1P19V1);  /*link up vert edges of room 530C*/
add_edge(H1P1V34a,H1P19V2);
add_edge(H1P1V34b,H1P50V1);
add_edge(H1P1V34c,H1P50V2);
add_edge(H1P1V34d,H1P50V3);
add_edge(H1P1V34e,H1P50V4);
add_edge(H1P1V34f,H1P19V3);
add_edge(H1P1V35,H1P19V4);
add_edge(H1P1V36,H1P2V2);  /*corner*/
add_edge(H1P1V37,H1P2V3);   /*corner*/
add_edge(H1P1V38,H1P20V1);  /*link up vert edges of room 421*/
add_edge(H1P1V38a,H1P20V2);
add_edge(H1P1V38b,H1P51V1);
add_edge(H1P1V38c,H1P51V2);
add_edge(H1P1V38d,H1P51V3);
add_edge(H1P1V38e,H1P51V4);
add_edge(H1P1V38f,H1P20V3);
add_edge(H1P1V39,H1P20V4);
add_edge(H1P1V40,H1P21V1);  /*link up vert edges of room 531*/
add_edge(H1P1V40a,H1P21V2);
add_edge(H1P1V40b,H1P52V1);
add_edge(H1P1V40c,H1P52V2);
add_edge(H1P1V40d,H1P52V3);
add_edge(H1P1V40e,H1P52V4);
add_edge(H1P1V40f,H1P21V3);
add_edge(H1P1V41,H1P21V4);
add_edge(H1P1V42,H1P22V1);  /*link up vert edges of room 529*/
add_edge(H1P1V42a,H1P22V2);
add_edge(H1P1V42b,H1P53V1);
add_edge(H1P1V42c,H1P53V2);
add_edge(H1P1V42d,H1P53V3);
add_edge(H1P1V42e,H1P53V4);
add_edge(H1P1V42f,H1P22V3);
add_edge(H1P1V43,H1P22V4);
add_edge(H1P1V44,H1P23V1);  /*link up vert edges of room 527*/
add_edge(H1P1V44a,H1P23V2);
add_edge(H1P1V44b,H1P54V1);
add_edge(H1P1V44c,H1P54V2);
add_edge(H1P1V44d,H1P54V3);
add_edge(H1P1V44e,H1P54V4);
add_edge(H1P1V44f,H1P23V3);
add_edge(H1P1V45,H1P23V4);
```

```
add_edge(H1P1V46,H1P24V1);  /*link up vert edges of room 525*/
add_edge(H1P1V46a,H1P24V2);
add_edge(H1P1V46b,H1P55V1);
add_edge(H1P1V46c,H1P55V2);
add_edge(H1P1V46d,H1P55V3);
add_edge(H1P1V46e,H1P55V4);
add_edge(H1P1V46f,H1P24V3);
add_edge(H1P1V47,H1P24V4);
add_edge(H1P1V48,H1P25V1);  /*link up vert edges of room 523*/
add_edge(H1P1V48a,H1P25V2);
add_edge(H1P1V48b,H1P56V1);
add_edge(H1P1V48c,H1P56V2);
add_edge(H1P1V48d,H1P56V3);
add_edge(H1P1V48e,H1P56V4);
add_edge(H1P1V48f,H1P25V3);
add_edge(H1P1V49,H1P25V4);
add_edge(H1P1V50,H1P26V1);  /*link up vert edges of room 521*/
add_edge(H1P1V50a,H1P26V2);
add_edge(H1P1V50b,H1P57V1);
add_edge(H1P1V50c,H1P57V2);
add_edge(H1P1V50d,H1P57V3);
add_edge(H1P1V50e,H1P57V4);
add_edge(H1P1V50f,H1P26V3);
add_edge(H1P1V51,H1P26V4);
add_edge(H1P1V52,H1P27V1);  /*link up vert edges of room 519*/
add_edge(H1P1V52a,H1P27V2);
add_edge(H1P1V52b,H1P58V1);
add_edge(H1P1V52c,H1P58V2);
add_edge(H1P1V52d,H1P58V3);
add_edge(H1P1V52e,H1P58V4);
add_edge(H1P1V52f,H1P27V3);
add_edge(H1P1V53,H1P27V4);
add_edge(H1P1V54,H1P3V2);  /*corner*/
add_edge(H1P1V55,H1P28V1);  /*link up vert edges of room ?*/
add_edge(H1P1V55a,H1P28V2);
add_edge(H1P1V55b,H1P59V1);
add_edge(H1P1V55c,H1P59V2);
add_edge(H1P1V55d,H1P59V3);
add_edge(H1P1V55e,H1P59V4);
add_edge(H1P1V55f,H1P28V3);
add_edge(H1P1V56,H1P28V4);
add_edge(H1P1V57,H1P3V3);  /*corner*/
add_edge(H1P1V58,H1P29V1);  /*link up vert edges of office 515 */
add_edge(H1P1V58a,H1P29V2);
add_edge(H1P1V58b,H1P60V1);
add_edge(H1P1V58c,H1P60V2);
add_edge(H1P1V58d,H1P60V3);
add_edge(H1P1V58e,H1P60V4);
add_edge(H1P1V58f,H1P29V3);
add_edge(H1P1V59,H1P29V4);
add_edge(H1P1V60,H1P30V1);  /*link up vert edges of room 511 */
add_edge(H1P1V60a,H1P30V2);
add_edge(H1P1V60b,H1P61V1);
add_edge(H1P1V60c,H1P61V2);
add_edge(H1P1V60d,H1P61V3);
add_edge(H1P1V60e,H1P61V4);
add_edge(H1P1V60f,H1P30V3);
add_edge(H1P1V61,H1P30V4);
add_edge(H1P1V62,H1P3V4);  /*corner*/


add_edge(H1P1V63,H1P63V1);  /*link up vert edges of room elev 1*/
```

```
add_edge(H1P1V63a,H1P63V2);
add_edge(H1P1V63b,H1P31V1);
add_edge(H1P1V63c,H1P31V2);
add_edge(H1P1V63d,H1P31V3);
add_edge(H1P1V63e,H1P31V4);
add_edge(H1P1V63f,H1P31V5);
add_edge(H1P1V63g,H1P63V3);
add_edge(H1P1V64,H1P63V4);
add_edge(H1P1V65,H1P64V1);  /*link up vert edges of room elev 2*/
add_edge(H1P1V65a,H1P64V2);
add_edge(H1P1V65b,H1P32V1);
add_edge(H1P1V65c,H1P32V2);
add_edge(H1P1V65d,H1P32V3);
add_edge(H1P1V65e,H1P32V4);
add_edge(H1P1V65f,H1P32V5);
add_edge(H1P1V65g,H1P64V3);
add_edge(H1P1V66,H1P64V4);

add_edge(H1P1V67,H1P3V1);  /*corner*/
add_edge(H1P1V68,H1P33V1);  /*link up vert edges of stairwell*/
add_edge(H1P1V68a,H1P33V2);
add_edge(H1P1V68b,H1P62V1);
add_edge(H1P1V68c,H1P62V2);
add_edge(H1P1V68d,H1P62V3);
add_edge(H1P1V68e,H1P62V4);
add_edge(H1P1V68f,H1P33V3);
add_edge(H1P1V69,H1P33V4);
add_edge(H1P1V70,H1P2V4);  /*corner*/



/* Now define the different classes of doors and put instances inside the
   door jams */

add_instance("hallway",7,H1,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0);

  H2=add_ph("office_door",11,W,0,1);
  H2P1=add_pg(H2,0.0,1,1);
  H2P1V1 = add_vertex(H2P1,0.0,0.0);
  H2P1V2 = add_vertex(H2P1,1.75,0.0);
  H2P1V3 = add_vertex(H2P1,1.75,35.5);
  H2P1V4 = add_vertex(H2P1,0.0,35.5);
  H2P2=add_pg(H2,83.5,0,1);
  H2P2V1 = add_vertex(H2P2,0.0,0.0);
  H2P2V2 = add_vertex(H2P2,1.75,0.0);
  H2P2V3 = add_vertex(H2P2,1.75,35.5);
  H2P2V4 = add_vertex(H2P2,0.0,35.5);
  add_edge(H2P1V1,H2P2V1);  /*link up vert edges of door*/
  add_edge(H2P1V2,H2P2V2);
  add_edge(H2P1V3,H2P2V3);
  add_edge(H2P1V4,H2P2V4);

  add_ceiling(H2P1,H2P2);

add_instance("door506",7,H2,-5.3,239.6,0.2,0.0,0.0,0.0);
add_instance("door510",7,H2,-5.3,713.8,0.2,0.0,0.0,0.0);
add_instance("door512",7,H2,-5.3,861.5,0.2,0.0,35.5,0.0);
add_instance("door514",7,H2,-5.3,937.6,0.2,0.0,0.0,0.0);
add_instance("door516",7,H2,-5.3,1085.3,0.2,0.0,35.5,0.0);
add_instance("door518",7,H2,-5.3,1161.8,0.2,0.0,0.0,0.0);
add_instance("door520",7,H2,-5.3,1309.0,0.2,0.0,35.5,0.0);
add_instance("door524",7,H2,-5.3,1618.9,0.2,0.0,35.5,0.0);
```

```
add_instance("door528A",8,H2,-5.3,1836.5,0.2,0.0,0.0,0.0);
add_instance("door528B",8,H2,-5.3,1919.2,0.2,0.0,0.0,0.0);
add_instance("door530A",8,H2,-5.3,2030.5,0.2,0.0,0.0,0.0);
add_instance("door530B",8,H2,-5.3,2230.7,0.2,0.0,35.5,0.0);
add_instance("door530C",8,H2,-5.3,2253.9,0.2,0.0,0.0,0.0);
add_instance("door421",7,H2,103.3,2136.3,0.2,1.75,0.0,0.0);
add_instance("door525",7,H2,103.3,1342.6,0.2,1.75,35.5,0.0);
add_instance("door523",7,H2,103.3,1118.7,0.2,1.75,35.5,0.0);
add_instance("door521",7,H2,103.3,796.0,0.2,1.75,35.5,0.0);
add_instance("door519",7,H2,103.3,564.4,0.2,1.75,35.5,0.0);
add_instance("door?",5,H2,293.8,415.65,0.2,0.0,35.5,90.0);
add_instance("door511",7,H2,342.8,310.1,0.2,1.75,35.5,0.0);
add_instance("doorstairs",10,H2,103.3,64.4,0.2,1.75,0.0,0.0);

  H3=add_ph("fire_door",9,W,0,1);
  H3P1=add_pg(H3,0.0,1,1);
  H3P1V1 = add_vertex(H3P1,0.0,0.0);
  H3P1V2 = add_vertex(H3P1,1.75,0.0);
  H3P1V3 = add_vertex(H3P1,1.75,35.6);
  H3P1V4 = add_vertex(H3P1,0.0,35.6);
  H3P2=add_pg(H3,82.9,0,1);
  H3P2V1 = add_vertex(H3P2,0.0,0.0);
  H3P2V2 = add_vertex(H3P2,1.75,0.0);
  H3P2V3 = add_vertex(H3P2,1.75,35.6);
  H3P2V4 = add_vertex(H3P2,0.0,35.6);
  add_edge(H3P1V1,H3P2V1);  /*link up vert edges of door*/
  add_edge(H3P1V2,H3P2V2);
  add_edge(H3P1V3,H3P2V3);
  add_edge(H3P1V4,H3P2V4);

  add_ceiling(H3P1,H3P2);

add_instance("1st_fire_door1",13,H3,-5.5,1488.3,0.2,0.0,0.0,0.0);
add_instance("1st_fire_door2",13,H3,-5.5,1559.7,0.2,0.0,35.6,0.0);


  H4=add_ph("restroom_door",13,W,0,1);
  H4P1=add_pg(H4,0.0,1,1);
  H4P1V1 = add_vertex(H4P1,0.0,0.0);
  H4P1V2 = add_vertex(H4P1,1.75,0.0);
  H4P1V3 = add_vertex(H4P1,1.75,31.5);
  H4P1V4 = add_vertex(H4P1,0.0,31.5);
  H4P2=add_pg(H4,83.25,0,1);
  H4P2V1 = add_vertex(H4P2,0.0,0.0);
  H4P2V2 = add_vertex(H4P2,1.75,0.0);
  H4P2V3 = add_vertex(H4P2,1.75,31.5);
  H4P2V4 = add_vertex(H4P2,0.0,31.5);
  add_edge(H4P1V1,H4P2V1);
  add_edge(H4P1V2,H4P2V2);
  add_edge(H4P1V3,H4P2V3);
  add_edge(H4P1V4,H4P2V4);

  add_ceiling(H4P1,H4P2);

add_instance("door522R",8,H4,-5.3,1461.0,0.2,0.0,31.5,0.0);
add_instance("door526R",8,H4,-5.3,1785.9,0.2,0.0,31.5,0.0);

  H5=add_ph("double_door",11,W,0,1);
  H5P1=add_pg(H5,0.0,1,1);
  H5P1V1 = add_vertex(H5P1,0.0,0.0);
  H5P1V2 = add_vertex(H5P1,1.75,0.0);
  H5P1V3 = add_vertex(H5P1,1.75,29.6);
```

```
  H5P1V4 = add_vertex(H5P1,0.0,29.6);
  H5P2=add_pg(H5,82.9,0,1);
  H5P2V1 = add_vertex(H5P2,0.0,0.0);
  H5P2V2 = add_vertex(H5P2,1.75,0.0);
  H5P2V3 = add_vertex(H5P2,1.75,29.6);
  H5P2V4 = add_vertex(H5P2,0.0,29.6);
  add_edge(H5P1V1,H5P2V1);  /*link up vert edges of door*/
  add_edge(H5P1V2,H5P2V2);
  add_edge(H5P1V3,H5P2V3);
  add_edge(H5P1V4,H5P2V4);

  add_ceiling(H5P1,H5P2);

add_instance("1door531",8,H5,103.3,1937.4,0.2,1.75,29.6,0.0);
add_instance("2door531",8,H5,103.3,1878.0,0.2,1.75,0.0,0.0);
add_instance("1door529",8,H5,103.3,1744.2,0.2,1.75,29.6,0.0);
add_instance("2door529",8,H5,103.3,1684.8,0.2,1.75,0.0,0.0);
add_instance("1door527",8,H5,103.3,1522.1,0.2,1.75,29.6,0.0);
add_instance("2door527",8,H5,103.3,1462.7,0.2,1.75,0.0,0.0);
add_instance("1door_office",12,H5,339.25,402.3,0.2,1.75,29.6,0.0);
add_instance("2door_office",12,H5,339.25,342.9,0.2,1.75,0.0,0.0);


/*   Notice that lights have no height */

  H6=add_ph("light",5,W,1,1);
  H6P1=add_pg(H6,0.0,1,1);
  H6P1V1 = add_vertex(H6P1,0.0,0.0);
  H6P1V2 = add_vertex(H6P1,45.5,0.0);
  H6P1V3 = add_vertex(H6P1,45.5,21.25);
  H6P1V4 = add_vertex(H6P1,0.0,21.25);

add_instance("light1",6,H6,26.25,98.5,102.0,0.0,0.0,0.0,0.0);
add_instance("light2",6,H6,26.25,362.75,102.0,0.0,0.0,0.0,0.0);
add_instance("light3",6,H6,26.25,651.0,102.0,0.0,0.0,0.0,0.0);
add_instance("light4",6,H6,26.25,915.25,102.0,0.0,0.0,0.0,0.0);
add_instance("light5",6,H6,26.25,1251.5,102.0,0.0,0.0,0.0,0.0);
add_instance("light6",6,H6,26.25,1539.75,102.0,0.0,0.0,0.0,0.0);
add_instance("light7",6,H6,26.25,1828.0,102.0,0.0,0.0,0.0,0.0);
add_instance("light8",6,H6,26.25,2140.25,102.0,0.0,0.0,0.0,0.0);

/* Since all molding sizes are different, we need to add a separate
   polyhedron for each one. But we still need to add one instance
   of each so it will appear in the model*/

/* 37 different molding pieces */

  H7=add_ph("molding1",8,W,1,1);
  H7P1=add_pg(H7,0.0,1,1);
  H7P1V1 = add_vertex(H7P1,0.0,0.0);
  H7P1V2 = add_vertex(H7P1,0.2,0.0);
  H7P1V3 = add_vertex(H7P1,0.2,237.5);
  H7P1V4 = add_vertex(H7P1,0.0,237.5);
  H7P2=add_pg(H7,3.875,0,1);
  H7P2V1 = add_vertex(H7P2,0.0,0.0);
  H7P2V2 = add_vertex(H7P2,0.2,0.0);
  H7P2V3 = add_vertex(H7P2,0.2,237.5);
  H7P2V4 = add_vertex(H7P2,0.0,237.5);
  add_edge(H7P1V1,H7P2V1);
  add_edge(H7P1V2,H7P2V2);
  add_edge(H7P1V3,H7P2V3);
  add_edge(H7P1V4,H7P2V4);
```

```
  add_ceiling(H7P1,H7P2);

add_instance("molding1",8,H7,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0);

  H8=add_ph("molding2",8,W,1,1);
  H8P1=add_pg(H8,0.0,1,1);
  H8P1V1 = add_vertex(H8P1,0.0,0.0);
  H8P1V2 = add_vertex(H8P1,0.2,0.0);
  H8P1V3 = add_vertex(H8P1,0.2,434.5);
  H8P1V4 = add_vertex(H8P1,0.0,434.5);
  H8P2=add_pg(H8,3.875,0,1);
  H8P2V1 = add_vertex(H8P2,0.0,0.0);
  H8P2V2 = add_vertex(H8P2,0.2,0.0);
  H8P2V3 = add_vertex(H8P2,0.2,434.5);
  H8P2V4 = add_vertex(H8P2,0.0,434.5);
  add_edge(H8P1V1,H8P2V1);
  add_edge(H8P1V2,H8P2V2);
  add_edge(H8P1V3,H8P2V3);
  add_edge(H8P1V4,H8P2V4);
  add_ceiling(H8P1,H8P2);

add_instance("molding2",8,H8,0.0,277.2,0.0,0.0,0.0,0.0,0.0);

  H9=add_ph("molding3",8,W,1,1);
  H9P1=add_pg(H9,0.0,1,1);
  H9P1V1 = add_vertex(H9P1,0.0,0.0);
  H9P1V2 = add_vertex(H9P1,0.2,0.0);
  H9P1V3 = add_vertex(H9P1,0.2,72.5);
  H9P1V4 = add_vertex(H9P1,0.0,72.5);
  H9P2=add_pg(H9,3.875,0,1);
  H9P2V1 = add_vertex(H9P2,0.0,0.0);
  H9P2V2 = add_vertex(H9P2,0.2,0.0);
  H9P2V3 = add_vertex(H9P2,0.2,72.5);
  H9P2V4 = add_vertex(H9P2,0.0,72.5);
  add_edge(H9P1V1,H9P2V1);
  add_edge(H9P1V2,H9P2V2);
  add_edge(H9P1V3,H9P2V3);
  add_edge(H9P1V4,H9P2V4);
  add_ceiling(H9P1,H9P2);

add_instance("molding3",8,H9,0.0,751.4,0.0,0.0,0.0,0.0,0.0);

  H10=add_ph("molding4",8,W,1,1);
  H10P1=add_pg(H10,0.0,1,1);
  H10P1V1 = add_vertex(H10P1,0.0,0.0);
  H10P1V2 = add_vertex(H10P1,0.2,0.0);
  H10P1V3 = add_vertex(H10P1,0.2,71.9);
  H10P1V4 = add_vertex(H10P1,0.0,71.9);
  H10P2=add_pg(H10,3.875,0,1);
  H10P2V1 = add_vertex(H10P2,0.0,0.0);
  H10P2V2 = add_vertex(H10P2,0.2,0.0);
  H10P2V3 = add_vertex(H10P2,0.2,71.9);
  H10P2V4 = add_vertex(H10P2,0.0,71.9);
  add_edge(H10P1V1,H10P2V1);
  add_edge(H10P1V2,H10P2V2);
  add_edge(H10P1V3,H10P2V3);
  add_edge(H10P1V4,H10P2V4);
  add_ceiling(H10P1,H10P2);

add_instance("molding4",8,H10,0.0,863.6,0.0,0.0,0.0,0.0,0.0);

  H11=add_ph("molding5",8,W,1,1);
```

```
  H11P1=add_pg(H11,0.0,1,1);
  H11P1V1 = add_vertex(H11P1,0.0,0.0);
  H11P1V2 = add_vertex(H11P1,0.2,0.0);
  H11P1V3 = add_vertex(H11P1,0.2,72.5);
  H11P1V4 = add_vertex(H11P1,0.0,72.5);
  H11P2=add_pg(H11,3.875,0,1);
  H11P2V1 = add_vertex(H11P2,0.0,0.0);
  H11P2V2 = add_vertex(H11P2,0.2,0.0);
  H11P2V3 = add_vertex(H11P2,0.2,72.5);
  H11P2V4 = add_vertex(H11P2,0.0,72.5);
  add_edge(H11P1V1,H11P2V1);
  add_edge(H11P1V2,H11P2V2);
  add_edge(H11P1V3,H11P2V3);
  add_edge(H11P1V4,H11P2V4);
  add_ceiling(H11P1,H11P2);

add_instance("molding5",8,H11,0.0,975.2,0.0,0.0,0.0,0.0,0.0);

  H12=add_ph("molding6",8,W,1,1);
  H12P1=add_pg(H12,0.0,1,1);
  H12P1V1 = add_vertex(H12P1,0.0,0.0);
  H12P1V2 = add_vertex(H12P1,0.2,0.0);
  H12P1V3 = add_vertex(H12P1,0.2,72.3);
  H12P1V4 = add_vertex(H12P1,0.0,72.3);
  H12P2=add_pg(H12,3.875,0,1);
  H12P2V1 = add_vertex(H12P2,0.0,0.0);
  H12P2V2 = add_vertex(H12P2,0.2,0.0);
  H12P2V3 = add_vertex(H12P2,0.2,72.3);
  H12P2V4 = add_vertex(H12P2,0.0,72.3);
  add_edge(H12P1V1,H12P2V1);
  add_edge(H12P1V2,H12P2V2);
  add_edge(H12P1V3,H12P2V3);
  add_edge(H12P1V4,H12P2V4);
  add_ceiling(H12P1,H12P2);

add_instance("molding6",8,H12,0.0,1087.4,0.0,0.0,0.0,0.0,0.0);

  H13=add_ph("molding7",8,W,1,1);
  H13P1=add_pg(H13,0.0,1,1);
  H13P1V1 = add_vertex(H13P1,0.0,0.0);
  H13P1V2 = add_vertex(H13P1,0.2,0.0);
  H13P1V3 = add_vertex(H13P1,0.2,72.0);
  H13P1V4 = add_vertex(H13P1,0.0,72.0);
  H13P2=add_pg(H13,3.875,0,1);
  H13P2V1 = add_vertex(H13P2,0.0,0.0);
  H13P2V2 = add_vertex(H13P2,0.2,0.0);
  H13P2V3 = add_vertex(H13P2,0.2,72.0);
  H13P2V4 = add_vertex(H13P2,0.0,72.0);
  add_edge(H13P1V1,H13P2V1);
  add_edge(H13P1V2,H13P2V2);
  add_edge(H13P1V3,H13P2V3);
  add_edge(H13P1V4,H13P2V4);
  add_ceiling(H13P1,H13P2);

add_instance("molding7",8,H13,0.0,1199.4,0.0,0.0,0.0,0.0,0.0);

  H14=add_ph("molding8",8,W,1,1);
  H14P1=add_pg(H14,0.0,1,1);
  H14P1V1 = add_vertex(H14P1,0.0,0.0);
  H14P1V2 = add_vertex(H14P1,0.2,0.0);
  H14P1V3 = add_vertex(H14P1,0.2,116.5);
  H14P1V4 = add_vertex(H14P1,0.0,116.5);
```

```
  H14P2=add_pg(H14,3.875,0.1);
  H14P2V1 = add_vertex(H14P2,0.0,0.0);
  H14P2V2 = add_vertex(H14P2,0.2,0.0);
  H14P2V3 = add_vertex(H14P2,0.2,116.5);
  H14P2V4 = add_vertex(H14P2,0.0,116.5);
  add_edge(H14P1V1,H14P2V1);
  add_edge(H14P1V2,H14P2V2);
  add_edge(H14P1V3,H14P2V3);
  add_edge(H14P1V4,H14P2V4);
  add_ceiling(H14P1,H14P2);

add_instance("molding8",8,H14,0.0,1311.1,0.0,0.0,0.0,0.0);

  H15=add_ph("molding9",8,W,1,1);
  H15P1=add_pg(H15,0.0,1,1);
  H15P1V1 = add_vertex(H15P1,0.0,0.0);
  H15P1V2 = add_vertex(H15P1,0.2,0.0);
  H15P1V3 = add_vertex(H15P1,0.2,22.7);
  H15P1V4 = add_vertex(H15P1,0.0,22.7);
  H15P2=add_pg(H15,3.875,0,1);
  H15P2V1 = add_vertex(H15P2,0.0,0.0);
  H15P2V2 = add_vertex(H15P2,0.2,0.0);
  H15P2V3 = add_vertex(H15P2,0.2,22.7);
  H15P2V4 = add_vertex(H15P2,0.0,22.7);
  add_edge(H15P1V1,H15P2V1);
  add_edge(H15P1V2,H15P2V2);
  add_edge(H15P1V3,H15P2V3);
  add_edge(H15P1V4,H15P2V4);
  add_ceiling(H15P1,H15P2);

add_instance("molding9",8,H15,0.0,1463.3,0.0,0.0,0.0,0.0,0.0);

  H16=add_ph("molding10",9,W,1,1);
  H16P1=add_pg(H16,0.0,1,1);
  H16P1V1 = add_vertex(H16P1,0.0,0.0);
  H16P1V2 = add_vertex(H16P1,0.2,0.0);
  H16P1V3 = add_vertex(H16P1,0.2,19.3);
  H16P1V4 = add_vertex(H16P1,0.0,19.3);
  H16P2=add_pg(H16,3.875,0,1);
  H16P2V1 = add_vertex(H16P2,0.0,0.0);
  H16P2V2 = add_vertex(H16P2,0.2,0.0);
  H16P2V3 = add_vertex(H16P2,0.2,19.3);
  H16P2V4 = add_vertex(H16P2,0.0,19.3);
  add_edge(H16P1V1,H16P2V1);
  add_edge(H16P1V2,H16P2V2);
  add_edge(H16P1V3,H16P2V3);
  add_edge(H16P1V4,H16P2V4);
  add_ceiling(H16P1,H16P2);

add_instance("molding10",9,H16,0.0,1562.0,0.0,0.0,0.0,0.0,0.0);

  H17=add_ph("molding11",9,W,1,1);
  H17P1=add_pg(H17,0.0,1,1);
  H17P1V1 = add_vertex(H17P1,0.0,0.0);
  H17P1V2 = add_vertex(H17P1,0.2,0.0);
  H17P1V3 = add_vertex(H17P1,0.2,31.4);
  H17P1V4 = add_vertex(H17P1,0.0,31.4);
  H17P2=add_pg(H17,3.875,0,1);
  H17P2V1 = add_vertex(H17P2,0.0,0.0);
  H17P2V2 = add_vertex(H17P2,0.2,0.0);
  H17P2V3 = add_vertex(H17P2,0.2,31.4);
  H17P2V4 = add_vertex(H17P2,0.0,31.4);
```

```
  add_edge(H17P1V1,H17P2V1);
  add_edge(H17P1V2,H17P2V2);
  add_edge(H17P1V3,H17P2V3);
  add_edge(H17P1V4,H17P2V4);
  add_ceiling(H17P1,H17P2);

add_instance("molding11",9,H17,0.0,1619.0,0.0,0.0,0.0,0.0,0.0);

  H18=add_ph("molding12",9,W,1,1);
  H18P1=add_pg(H18,0.0,1,1);
  H18P1V1 = add_vertex(H18P1,0.0,0.0);
  H18P1V2 = add_vertex(H18P1,0.2,0.0);
  H18P1V3 = add_vertex(H18P1,0.2,68.0);
  H18P1V4 = add_vertex(H18P1,0.0,68.0);
  H18P2=add_pg(H18,3.875,0,1);
  H18P2V1 = add_vertex(H18P2,0.0,0.0);
  H18P2V2 = add_vertex(H18P2,0.2,0.0);
  H18P2V3 = add_vertex(H18P2,0.2,68.0);
  H18P2V4 = add_vertex(H18P2,0.0,68.0);
  add_edge(H18P1V1,H18P2V1);
  add_edge(H18P1V2,H18P2V2);
  add_edge(H18P1V3,H18P2V3);
  add_edge(H18P1V4,H18P2V4);
  add_ceiling(H18P1,H18P2);

add_instance("molding12",9,H18,0.0,1684.5,0.0,0.0,0.0,0.0,0.0);

  H19=add_ph("molding13",9,W,1,1);
  H19P1=add_pg(H19,0.0,1,1);
  H19P1V1 = add_vertex(H19P1,0.0,0.0);
  H19P1V2 = add_vertex(H19P1,0.2,0.0);
  H19P1V3 = add_vertex(H19P1,0.2,46.2);
  H19P1V4 = add_vertex(H19P1,0.0,46.2);
  H19P2=add_pg(H19,3.875,0,1);
  H19P2V1 = add_vertex(H19P2,0.0,0.0);
  H19P2V2 = add_vertex(H19P2,0.2,0.0);
  H19P2V3 = add_vertex(H19P2,0.2,46.2);
  H19P2V4 = add_vertex(H19P2,0.0,46.2);
  add_edge(H19P1V1,H19P2V1);
  add_edge(H19P1V2,H19P2V2);
  add_edge(H19P1V3,H19P2V3);
  add_edge(H19P1V4,H19P2V4);
  add_ceiling(H19P1,H19P2);

add_instance("molding13",9,H19,0.0,1788.2,0.0,0.0,0.0,0.0,0.0);

  H20=add_ph("molding14",9,W,1,1);
  H20P1=add_pg(H20,0.0,1,1);
  H20P1V1 = add_vertex(H20P1,0.0,0.0);
  H20P1V2 = add_vertex(H20P1,0.2,0.0);
  H20P1V3 = add_vertex(H20P1,0.2,43.0);
  H20P1V4 = add_vertex(H20P1,0.0,43.0);
  H20P2=add_pg(H20,3.875,0,1);
  H20P2V1 = add_vertex(H20P2,0.0,0.0);
  H20P2V2 = add_vertex(H20P2,0.2,0.0);
  H20P2V3 = add_vertex(H20P2,0.2,43.0);
  H20P2V4 = add_vertex(H20P2,0.0,43.0);
  add_edge(H20P1V1,H20P2V1);
  add_edge(H20P1V2,H20P2V2);
  add_edge(H20P1V3,H20P2V3);
  add_edge(H20P1V4,H20P2V4);
  add_ceiling(H20P1,H20P2);
```

```
add_instance("molding14",9,H20,0.0,1874.1,0.0,0.0,0.0,0.0,0.0);

  H21=add_ph("molding15",9,W,1,1);
  H21P1=add_pg(H21,0.0,1,1);
  H21P1V1 = add_vertex(H21P1,0.0,0.0);
  H21P1V2 = add_vertex(H21P1,0.2,0.0);
  H21P1V3 = add_vertex(H21P1,0.2,71.6);
  H21P1V4 = add_vertex(H21P1,0.0,71.6);
  H21P2=add_pg(H21,3.875,0,1);
  H21P2V1 = add_vertex(H21P2,0.0,0.0);
  H21P2V2 = add_vertex(H21P2,0.2,0.0);
  H21P2V3 = add_vertex(H21P2,0.2,71.6);
  H21P2V4 = add_vertex(H21P2,0.0,71.6);
  add_edge(H21P1V1,H21P2V1);
  add_edge(H21P1V2,H21P2V2);
  add_edge(H21P1V3,H21P2V3);
  add_edge(H21P1V4,H21P2V4);
  add_ceiling(H21P1,H21P2);

add_instance("molding15",9,H21,0.0,1956.8,0.0,0.0,0.0,0.0,0.0);

  H22=add_ph("molding16",9,W,1,1);
  H22P1=add_pg(H22,0.0,1,1);
  H22P1V1 = add_vertex(H22P1,0.0,0.0);
  H22P1V2 = add_vertex(H22P1,0.2,0.0);
  H22P1V3 = add_vertex(H22P1,0.2,125.0);
  H22P1V4 = add_vertex(H22P1,0.0,125.0);
  H22P2=add_pg(H22,3.875,0,1);
  H22P2V1 = add_vertex(H22P2,0.0,0.0);
  H22P2V2 = add_vertex(H22P2,0.2,0.0);
  H22P2V3 = add_vertex(H22P2,0.2,125.0);
  H22P2V4 = add_vertex(H22P2,0.0,125.0);
  add_edge(H22P1V1,H22P2V1);
  add_edge(H22P1V2,H22P2V2);
  add_edge(H22P1V3,H22P2V3);
  add_edge(H22P1V4,H22P2V4);
  add_ceiling(H22P1,H22P2);

add_instance("molding16",9,H22,0.0,2068.1,0.0,0.0,0.0,0.0,0.0);

  H23=add_ph("molding9",8,W,1,1);
  H23P1=add_pg(H23,0.0,1,1);
  H23P1V1 = add_vertex(H23P1,0.0,0.0);
  H23P1V2 = add_vertex(H23P1,0.2,0.0);
  H23P1V3 = add_vertex(H23P1,0.2,19.0);
  H23P1V4 = add_vertex(H23P1,0.0,19.0);
  H23P2=add_pg(H23,3.875,0,1);
  H23P2V1 = add_vertex(H23P2,0.0,0.0);
  H23P2V2 = add_vertex(H23P2,0.2,0.0);
  H23P2V3 = add_vertex(H23P2,0.2,19.0);
  H23P2V4 = add_vertex(H23P2,0.0,19.0);
  add_edge(H23P1V1,H23P2V1);
  add_edge(H23P1V2,H23P2V2);
  add_edge(H23P1V3,H23P2V3);
  add_edge(H23P1V4,H23P2V4);
  add_ceiling(H23P1,H23P2);

add_instance("molding16",9,H23,0.0,2232.8,0.0,0.0,0.0,0.0,0.0);

  H24=add_ph("molding17",9,W,1,1);
  H24P1=add_pg(H24,0.0,1,1);
  H24P1V1 = add_vertex(H24P1,0.0,0.0);
```

```
  H24P1V2 = add_vertex(H24P1,0.2,0.0);
  H24P1V3 = add_vertex(H24P1,0.2,61.7);
  H24P1V4 = add_vertex(H24P1,0.0,61.7);
  H24P2=add_pg(H24,3.875,0.1);
  H24P2V1 = add_vertex(H24P2,0.0,0.0);
  H24P2V2 = add_vertex(H24P2,0.2,0.0);
  H24P2V3 = add_vertex(H24P2,0.2,61.7);
  H24P2V4 = add_vertex(H24P2,0.0,61.7);
  add_edge(H24P1V1,H24P2V1);
  add_edge(H24P1V2,H24P2V2);
  add_edge(H24P1V3,H24P2V3);
  add_edge(H24P1V4,H24P2V4);
  add_ceiling(H24P1,H24P2);

add_instance("molding17",9,H24,0.0,2289.5,0.0,0.0,0.0,0.0,0.0);

  H25=add_ph("molding18",9,W,1,1);
  H25P1=add_pg(H25,0.0,1,1);
  H25P1V1 = add_vertex(H25P1,0.0,0.0);
  H25P1V2 = add_vertex(H25P1,0.0,177.3);
  H25P1V3 = add_vertex(H25P1,-0.2,177.3);
  H25P1V4 = add_vertex(H25P1,-0.2,0.0);
  H25P2=add_pg(H25,3.875,0.1);
  H25P2V1 = add_vertex(H25P2,0.0,0.0);
  H25P2V2 = add_vertex(H25P2,0.0,177.3);
  H25P2V3 = add_vertex(H25P2,-0.2,177.3);
  H25P2V4 = add_vertex(H25P2,-0.2,0.0);
  add_edge(H25P1V1,H25P2V1);
  add_edge(H25P1V2,H25P2V2);
  add_edge(H25P1V3,H25P2V3);
  add_edge(H25P1V4,H25P2V4);
  add_ceiling(H25P1,H25P2);

add_instance("molding18",9,H25,98.0,2173.9,0.0,0.0,0.0,0.0,0.0);

  H26=add_ph("molding19",9,W,1,1);
  H26P1=add_pg(H26,0.0,1,1);
  H26P1V1 = add_vertex(H26P1,0.0,0.0);
  H26P1V2 = add_vertex(H26P1,0.0,194.5);
  H26P1V3 = add_vertex(H26P1,-0.2,194.5);
  H26P1V4 = add_vertex(H26P1,-0.2,0.0);
  H26P2=add_pg(H26,3.875,0,1);
  H26P2V1 = add_vertex(H26P2,0.0,0.0);
  H26P2V2 = add_vertex(H26P2,0.0,194.5);
  H26P2V3 = add_vertex(H26P2,-0.2,194.5);
  H26P2V4 = add_vertex(H26P2,-0.2,0.0);
  add_edge(H26P1V1,H26P2V1);
  add_edge(H26P1V2,H26P2V2);
  add_edge(H26P1V3,H26P2V3);
  add_edge(H26P1V4,H26P2V4);
  add_ceiling(H26P1,H26P2);

add_instance("molding19",9,H26,98.0,1939.7,0.0,0.0,0.0,0.0,0.0);


  H27=add_ph("molding20",9,W,1,1);
  H27P1=add_pg(H27,0.0,1,1);
  H27P1V1 = add_vertex(H27P1,0.0,0.0);
  H27P1V2 = add_vertex(H27P1,0.0,129.2);
  H27P1V3 = add_vertex(H27P1,-0.2,129.2);
  H27P1V4 = add_vertex(H27P1,-0.2,0.0);
  H27P2=add_pg(H27,3.875,0,1);
```

```
  H27P2V1 = add_vertex(H27P2,0.0,0.0);
  H27P2V2 = add_vertex(H27P2,0.0,129.2);
  H27P2V3 = add_vertex(H27P2,-0.2,129.2);
  H27P2V4 = add_vertex(H27P2,-0.2,0.0);
  add_edge(H27P1V1,H27P2V1);
  add_edge(H27P1V2,H27P2V2);
  add_edge(H27P1V3,H27P2V3);
  add_edge(H27P1V4,H27P2V4);
  add_ceiling(H27P1,H27P2);

add_instance("molding20",9,H27,98.0,1746.5,0.0,0.0,0.0,0.0,0.0);

  H28=add_ph("molding21",9,W,1,1);
  H28P1=add_pg(H28,0.0,1,1);
  H28P1V1 = add_vertex(H28P1,0.0,0.0);
  H28P1V2 = add_vertex(H28P1,0.0,158.1);
  H28P1V3 = add_vertex(H28P1,-0.2,158.1);
  H28P1V4 = add_vertex(H28P1,-0.2,0.0);
  H28P2=add_pg(H28,3.875,0,1);
  H28P2V1 = add_vertex(H28P2,0.0,0.0);
  H28P2V2 = add_vertex(H28P2,0.0,158.1);
  H28P2V3 = add_vertex(H28P2,-0.2,158.1);
  H28P2V4 = add_vertex(H28P2,-0.2,0.0);
  add_edge(H28P1V1,H28P2V1);
  add_edge(H28P1V2,H28P2V2);
  add_edge(H28P1V3,H28P2V3);
  add_edge(H28P1V4,H28P2V4);
  add_ceiling(H28P1,H28P2);

add_instance("molding21",9,H28,98.0,1524.4,0.0,0.0,0.0,0.0,0.0);

  H29=add_ph("molding22",9,W,1,1);
  H29P1=add_pg(H29,0.0,1,1);
  H29P1V1 = add_vertex(H29P1,0.0,0.0);
  H29P1V2 = add_vertex(H29P1,0.0,115.7);
  H29P1V3 = add_vertex(H29P1,-0.2,115.7);
  H29P1V4 = add_vertex(H29P1,-0.2,0.0);
  H29P2=add_pg(H29,3.875,0,1);
  H29P2V1 = add_vertex(H29P2,0.0,0.0);
  H29P2V2 = add_vertex(H29P2,0.0,115.7);
  H29P2V3 = add_vertex(H29P2,-0.2,115.7);
  H29P2V4 = add_vertex(H29P2,-0.2,0.0);
  add_edge(H29P1V1,H29P2V1);
  add_edge(H29P1V2,H29P2V2);
  add_edge(H29P1V3,H29P2V3);
  add_edge(H29P1V4,H29P2V4);
  add_ceiling(H29P1,H29P2);

add_instance("molding22",9,H29,98.0,1344.7,0.0,0.0,0.0,0.0,0.0);

  H30=add_ph("molding23",9,W,1,1);
  H30P1=add_pg(H30,0.0,1,1);
  H30P1V1 = add_vertex(H30P1,0.0,0.0);
  H30P1V2 = add_vertex(H30P1,0.0,184.2);
  H30P1V3 = add_vertex(H30P1,-0.2,184.2);
  H30P1V4 = add_vertex(H30P1,-0.2,0.0);
  H30P2=add_pg(H30,3.875,0,1);
  H30P2V1 = add_vertex(H30P2,0.0,0.0);
  H30P2V2 = add_vertex(H30P2,0.0,184.2);
  H30P2V3 = add_vertex(H30P2,-0.2,184.2);
  H30P2V4 = add_vertex(H30P2,-0.2,0.0);
  add_edge(H30P1V1,H30P2V1);
```

```
  add_edge(H30P1V2,H30P2V2);
  add_edge(H30P1V3,H30P2V3);
  add_edge(H30P1V4,H30P2V4);
  add_ceiling(H30P1,H30P2);

add_instance("molding24",9,H30,98.0,1120.8,0.0,0.0,0.0,0.0,0.0);

  H31=add_ph("molding25",9,W,1,1);
  H31P1=add_pg(H31,0.0,1,1);
  H31P1V1 = add_vertex(H31P1,0.0,0.0);
  H31P1V2 = add_vertex(H31P1,0.0,283.0);
  H31P1V3 = add_vertex(H31P1,-0.2,283.0);
  H31P1V4 = add_vertex(H31P1,-0.2,0.0);
  H31P2=add_pg(H31,3.875,0,1);
  H31P2V1 = add_vertex(H31P2,0.0,0.0);
  H31P2V2 = add_vertex(H31P2,0.0,283.0);
  H31P2V3 = add_vertex(H31P2,-0.2,283.0);
  H31P2V4 = add_vertex(H31P2,-0.2,0.0);
  add_edge(H31P1V1,H31P2V1);
  add_edge(H31P1V2,H31P2V2);
  add_edge(H31P1V3,H31P2V3);
  add_edge(H31P1V4,H31P2V4);
  add_ceiling(H31P1,H31P2);

add_instance("molding25",9,H31,98.0,798.1,0.0,0.0,0.0,0.0,0.0);

  H32=add_ph("molding26",9,W,1,1);
  H32P1=add_pg(H32,0.0,1,1);
  H32P1V1 = add_vertex(H32P1,0.0,0.0);
  H32P1V2 = add_vertex(H32P1,0.0,191.9);
  H32P1V3 = add_vertex(H32P1,-0.2,191.9);
  H32P1V4 = add_vertex(H32P1,-0.2,0.0);
  H32P2=add_pg(H32,3.875,0,1);
  H32P2V1 = add_vertex(H32P2,0.0,0.0);
  H32P2V2 = add_vertex(H32P2,0.0,191.9);
  H32P2V3 = add_vertex(H32P2,-0.2,191.9);
  H32P2V4 = add_vertex(H32P2,-0.2,0.0);
  add_edge(H32P1V1,H32P2V1);
  add_edge(H32P1V2,H32P2V2);
  add_edge(H32P1V3,H32P2V3);
  add_edge(H32P1V4,H32P2V4);
  add_ceiling(H32P1,H32P2);

add_instance("molding26",9,H32,98.0,566.5,0.0,0.0,0.0,0.0,0.0);

  H33=add_ph("molding27",9,W,1,1);
  H33P1=add_pg(H33,0.0,1,1);
  H33P1V1 = add_vertex(H33P1,0.0,0.0);
  H33P1V2 = add_vertex(H33P1,0.0,112.9);
  H33P1V3 = add_vertex(H33P1,-0.2,112.9);
  H33P1V4 = add_vertex(H33P1,-0.2,0.0);
  H33P2=add_pg(H33,3.875,0,1);
  H33P2V1 = add_vertex(H33P2,0.0,0.0);
  H33P2V2 = add_vertex(H33P2,0.0,112.9);
  H33P2V3 = add_vertex(H33P2,-0.2,112.9);
  H33P2V4 = add_vertex(H33P2,-0.2,0.0);
  add_edge(H33P1V1,H33P2V1);
  add_edge(H33P1V2,H33P2V2);
  add_edge(H33P1V3,H33P2V3);
  add_edge(H33P1V4,H33P2V4);
  add_ceiling(H33P1,H33P2);
```

```
add_instance("molding27",9,H33,98.0,413.9,0.0,0.0,0.0,0.0);

  H34=add_ph("molding28",9,W,1,1);
  H34P1=add_pg(H34,0.0,1,1);
  H34P1V1 = add_vertex(H34P1,0.0,0.0);
  H34P1V2 = add_vertex(H34P1,0.0,-0.2);
  H34P1V3 = add_vertex(H34P1,157.9,-0.2);
  H34P1V4 = add_vertex(H34P1,157.9,0.0);
  H34P2=add_pg(H34,3.875,0,1);
  H34P2V1 = add_vertex(H34P2,0.0,0.0);
  H34P2V2 = add_vertex(H34P2,0.0,-0.2);
  H34P2V3 = add_vertex(H34P2,157.9,-0.2);
  H34P2V4 = add_vertex(H34P2,157.9,0.0);
  add_edge(H34P1V1,H34P2V1);
  add_edge(H34P1V2,H34P2V2);
  add_edge(H34P1V3,H34P2V3);
  add_edge(H34P1V4,H34P2V4);
  add_ceiling(H34P1,H34P2);

add_instance("molding28",9,H34,98.0,413.9,0.0,0.0,0.0,0.0);

  H35=add_ph("molding29",9,W,1,1);
  H35P1=add_pg(H35,0.0,1,1);
  H35P1V1 = add_vertex(H35P1,0.0,0.0);
  H35P1V2 = add_vertex(H35P1,0.0,-0.2);
  H35P1V3 = add_vertex(H35P1,41.6,-0.2);
  H35P1V4 = add_vertex(H35P1,41.6,0.0);
  H35P2=add_pg(H35,3.875,0,1);
  H35P2V1 = add_vertex(H35P2,0.0,0.0);
  H35P2V2 = add_vertex(H35P2,0.0,-0.2);
  H35P2V3 = add_vertex(H35P2,41.6,-0.2);
  H35P2V4 = add_vertex(H35P2,41.6,0.0);
  add_edge(H35P1V1,H35P2V1);
  add_edge(H35P1V2,H35P2V2);
  add_edge(H35P1V3,H35P2V3);
  add_edge(H35P1V4,H35P2V4);
  add_ceiling(H35P1,H35P2);

add_instance("molding29",9,H35,295.9,413.9,0.0,0.0,0.0,0.0);

  H36=add_ph("molding30",9,W,1,1);
  H36P1=add_pg(H36,0.0,1,1);
  H36P1V1 = add_vertex(H36P1,0.0,0.0);
  H36P1V2 = add_vertex(H36P1,-0.2,0.0);
  H36P1V3 = add_vertex(H36P1,-0.2,9.3);
  H36P1V4 = add_vertex(H36P1,0.0,9.3);
  H36P2=add_pg(H36,3.875,0,1);
  H36P2V1 = add_vertex(H36P2,0.0,0.0);
  H36P2V2 = add_vertex(H36P2,-0.2,0.0);
  H36P2V3 = add_vertex(H36P2,-0.2,9.3);
  H36P2V4 = add_vertex(H36P2,0.0,9.3);
  add_edge(H36P1V1,H36P2V1);
  add_edge(H36P1V2,H36P2V2);
  add_edge(H36P1V3,H36P2V3);
  add_edge(H36P1V4,H36P2V4);
  add_ceiling(H36P1,H36P2);

add_instance("molding30",9,H36,337.5,404.6,0.0,0.0,0.0,0.0);

  H37=add_ph("molding31",9,W,1,1);
  H37P1=add_pg(H37,0.0,1,1);
  H37P1V1 = add_vertex(H37P1,0.0,0.0);
```

```
  H37P1V2 = add_vertex(H37P1,-0.2,0.0);
  H37P1V3 = add_vertex(H37P1,-0.2,28.4);
  H37P1V4 = add_vertex(H37P1,0.0,28.4);
  H37P2=add_pg(H37,3.875,0,1);
  H37P2V1 = add_vertex(H37P2,0.0,0.0);
  H37P2V2 = add_vertex(H37P2,-0.2,0.0);
  H37P2V3 = add_vertex(H37P2,-0.2,28.4);
  H37P2V4 = add_vertex(H37P2,0.0,28.4);
  add_edge(H37P1V1,H37P2V1);
  add_edge(H37P1V2,H37P2V2);
  add_edge(H37P1V3,H37P2V3);
  add_edge(H37P1V4,H37P2V4);
  add_ceiling(H37P1,H37P2);

add_instance("molding31",9,H37,337.5,312.2,0.0,0.0,0.0,0.0);

  H38=add_ph("molding32",9,W,1,1);
  H38P1=add_pg(H38,0.0,1,1);
  H38P1V1 = add_vertex(H38P1,0.0,0.0);
  H38P1V2 = add_vertex(H38P1,-0.2,0.0);
  H38P1V3 = add_vertex(H38P1,-0.2,5.1);
  H38P1V4 = add_vertex(H38P1,0.0,5.1);
  H38P2=add_pg(H38,3.875,0,1);
  H38P2V1 = add_vertex(H38P2,0.0,0.0);
  H38P2V2 = add_vertex(H38P2,-0.2,0.0);
  H38P2V3 = add_vertex(H38P2,-0.2,5.1);
  H38P2V4 = add_vertex(H38P2,0.0,5.1);
  add_edge(H38P1V1,H38P2V1);
  add_edge(H38P1V2,H38P2V2);
  add_edge(H38P1V3,H38P2V3);
  add_edge(H38P1V4,H38P2V4);
  add_ceiling(H38P1,H38P2);

add_instance("molding32",9,H38,337.5,267.4,0.0,0.0,0.0,0.0);

  H39=add_ph("molding33",9,W,1,1);
  H39P1=add_pg(H39,0.0,1,1);
  H39P1V1 = add_vertex(H39P1,0.0,0.0);
  H39P1V2 = add_vertex(H39P1,30.6,0.0);
  H39P1V3 = add_vertex(H39P1,30.6,0.2);
  H39P1V4 = add_vertex(H39P1,0.0,0.2);
  H39P2=add_pg(H39,3.875,0,1);
  H39P2V1 = add_vertex(H39P2,0.0,0.0);
  H39P2V2 = add_vertex(H39P2,30.6,0.0);
  H39P2V3 = add_vertex(H39P2,30.6,0.2);
  H39P2V4 = add_vertex(H39P2,0.0,0.2);
  add_edge(H39P1V1,H39P2V1);
  add_edge(H39P1V2,H39P2V2);
  add_edge(H39P1V3,H39P2V3);
  add_edge(H39P1V4,H39P2V4);
  add_ceiling(H39P1,H39P2);

add_instance("molding33",9,H39,306.9,267.4,0.0,0.0,0.0,0.0);

  H40=add_ph("molding34",9,W,1,1);
  H40P1=add_pg(H40,0.0,1,1);
  H40P1V1 = add_vertex(H40P1,0.0,0.0);
  H40P1V2 = add_vertex(H40P1,56.7,0.0);
  H40P1V3 = add_vertex(H40P1,56.7,0.2);
  H40P1V4 = add_vertex(H40P1,0.0,0.2);
  H40P2=add_pg(H40,3.875,0,1);
  H40P2V1 = add_vertex(H40P2,0.0,0.0);
```

```
  H40P2V2 = add_vertex(H40P2,56.7,0.0);
  H40P2V3 = add_vertex(H40P2,56.7,0.2);
  H40P2V4 = add_vertex(H40P2,0.0,0.2);
  add_edge(H40P1V1,H40P2V1);
  add_edge(H40P1V2,H40P2V2);
  add_edge(H40P1V3,H40P2V3);
  add_edge(H40P1V4,H40P2V4);
  add_ceiling(H40P1,H40P2);

add_instance("molding34",9,H40,192.2,267.4,0.0,0.0,0.0,0.0,0.0);

  H41=add_ph("molding35",9,W,1,1);
  H41P1=add_pg(H41,0.0,1,1);
  H41P1V1 = add_vertex(H41P1,0.0,0.0);
  H41P1V2 = add_vertex(H41P1,36.2,0.0);
  H41P1V3 = add_vertex(H41P1,36.2,0.2);
  H41P1V4 = add_vertex(H41P1,0.0,0.2);
  H41P2=add_pg(H41,3.875,0,1);
  H41P2V1 = add_vertex(H41P2,0.0,0.0);
  H41P2V2 = add_vertex(H41P2,36.2,0.0);
  H41P2V3 = add_vertex(H41P2,36.2,0.2);
  H41P2V4 = add_vertex(H41P2,0.0,0.2);
  add_edge(H41P1V1,H41P2V1);
  add_edge(H41P1V2,H41P2V2);
  add_edge(H41P1V3,H41P2V3);
  add_edge(H41P1V4,H41P2V4);
  add_ceiling(H41P1,H41P2);

add_instance("molding35",9,H41,98.0,267.4,0.0,0.0,0.0,0.0,0.0);

  H42=add_ph("molding36",9,W,1,1);
  H42P1=add_pg(H42,0.0,1,1);
  H42P1V1 = add_vertex(H42P1,0.0,0.0);
  H42P1V2 = add_vertex(H42P1,0.0,-0.2);
  H42P1V3 = add_vertex(H42P1,165.4,-0.2);
  H42P1V4 = add_vertex(H42P1,165.4,0.0);
  H42P2=add_pg(H42,3.875,0,1);
  H42P2V1 = add_vertex(H42P2,0.0,0.0);
  H42P2V2 = add_vertex(H42P2,0.0,-0.2);
  H42P2V3 = add_vertex(H42P2,165.4,-0.2);
  H42P2V4 = add_vertex(H42P2,165.4,0.0);
  add_edge(H42P1V1,H42P2V1);
  add_edge(H42P1V2,H42P2V2);
  add_edge(H42P1V3,H42P2V3);
  add_edge(H42P1V4,H42P2V4);
  add_ceiling(H42P1,H42P2);

add_instance("molding36",9,H42,98.0,102.0,0.0,0.0,0.0,0.0,0.0);

  H43=add_ph("molding37",9,W,1,1);
  H43P1=add_pg(H43,0.0,1,1);
  H43P1V1 = add_vertex(H43P1,0.0,0.0);
  H43P1V2 = add_vertex(H43P1,-0.2,0.0);
  H43P1V3 = add_vertex(H43P1,-0.2,62.3);
  H43P1V4 = add_vertex(H43P1,0.0,62.3);
  H43P2=add_pg(H43,3.875,0,1);
  H43P2V1 = add_vertex(H43P2,0.0,0.0);
  H43P2V2 = add_vertex(H43P2,-0.2,0.0);
  H43P2V3 = add_vertex(H43P2,-0.2,62.3);
  H43P2V4 = add_vertex(H43P2,0.0,62.3);
  add_edge(H43P1V1,H43P2V1);
  add_edge(H43P1V2,H43P2V2);
```

```
  add_edge(H43P1V3,H43P2V3);
  add_edge(H43P1V4,H43P2V4);
  add_ceiling(H43P1,H43P2);

add_instance("molding37",9,H43,98.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0);

  return W;          /*return pointer to this entire world structure*/
}
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library 2
Code 52
Naval Postgraduate School
Monterey, CA 93943

Director of Research Administration 1
Code 012
Naval Postgraduate School
Monterey, CA 93943

Chairman, Code CS 2
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Yutaka Kanayam, Code CS/Ka 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Lt. James Stein 1
118 Brookside Rd.
Newtown Square, PA 19073

Thesis
S68254 Stein
c.1 Modelling, visibility testing and projection of an orthogonal three dimensional world in support of a single camera vision system.